Spécification d'une machine de gestion mémoire pour les interpréteurs des langages logiques.Version 1 (provisoire)

Yves Bekkers, Bernard Canet, Olivier Ridoux, Lucien Ungaro

IRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 283

SPÉCIFICATION D'UNE MACHINE DE GESTION MÉMOIRE POUR LES INTERPRÉTEURS DES LANGAGES LOGIQUES

Version 1 (provisoire)

Yves BEKKERS
Bernard CANET
Olivier RIDOUX
Lucien UNGARO

Avril 1984

IRISA — PUBLICATIONS INTERNES -

SPECIFICATION D'UNE MACHINE DE GESTION MEMOIRE POUR LES INTERPRETEURS DES LANGAGES LOGIQUES

VERSION 1 (PROVISOIRE)

Yves BEKKERS, Bernard CANET
Ollivier RIDOUX, Lucien UNGARO

Publication n° 222 - Février 1984

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Spécification d'une machine de gestion mémoire pour les interpréteurs des langages logiques - version 1 (provisoire)

Bekkers Y., Canet B., Ridoux O., Ungaro L.

Publication Interne n°222
82 pages

Février 1984

Résumé

Ce rapport propose une machine intermédiaire adaptée à l'interprétation des langages de programmation logique (PROLOG). Elle n'est pas un interpréteur complet d'un tel langage, mais plutôt une mémoire spécialisée, gouvernable par des commandes. Elle est adaptée, par sa structure et les services qu'elle offre, à la sorte de conservation d'information que nécessite un tel interpréteur. Elle s'acquitte de façon transparente des tâches délicates d'allocation et de récupération des ressources de mémoire, en prenant en compte l'indéterminisme, ce qui constitue une originalité par rapport à une machine LISP. Le traitement de ces problèmes à un niveau proche du matériel permet d'espérer de bonnes performances pour le programme interpréteur supporté par la machine.

Abstract

This report is a proposal for an intermediate machine which is well suited for interpreting logic progamming languages (PROLOG). This machine is not a complete interpreter of such a language. Instead it is a specialised memory, driven by commands. The services offered by the machine have been adapted to the kind of information recording needed by such interpreters. Memory allocation and garbage collection are realised transparently within the machine, taking into account the indeterminism introduced by logic programming languages, and therefore making the difference with classical LISP machines. Managing these problems at hardware level allows to expect good performances for the interpreter supported by the machine.

TABLE DES MATIERES

partie 1

PRESENTATION

La machine décrite dans ce document est destinée à l'interprétation d'une classe de langages de programmation née du besoin d'exprimer des traitements sur des structures formelles complexes et non plus essentiellement sur des données numériques. Parmi les domaines potentiellement concernés, on peut citer l'intelligence artificielle, la conception assistée, les systèmes experts et l'analyse et la compréhension des langues naturelles et artificielles.

Aux langages de traitements formels de type fonctionnel, tels que LISP, se sont ajoutés les langages "logiques", de type relationnel, basés historiquement sur le formalisme de la logique des prédicats, et dont l'originalité est la notion d'"indéterminisme". L'un de ceux-ci est le langage PROLOG, abréviation de PROgrammation LOGique. Ce langage est décrit dans de nombreux articles et ouvrages, notamment:

- **A machine-oriented logic based on the resolution principle** J.A. Robinson, journal of the ACM vol. 12 no 1 Janvier 1965.

- **Predicate logic as programming language** R.A. Kowalski, IFIP North Holland 1974, pages 569 à 574.

-**Sur les bases théoriques de PROLOG** A. Colmerauer, Groupe d'Intelligence Artificielle, Marseille Lumigny, 1979.

-**Etude et réalisation d'un système PROLOG** A. Colmerauer, H. Kanoui, M. Van Caneghem, Groupe d'Intelligence Artificielle, Marseille Lumigny, 1979.

- **PROLOG II manuel d'utilisation** M. Van Caneghem, Groupe d'Intelligence Artificielle, Marseille Lumigny, Janvier 1982.

La communauté scientifique et technique mondiale accorde une grande importance au développement industriel de machines capables de supporter eificacement la programmation en de tels langages. Les langages relationnels sont considérés actuellement, en 1983, comme un des axes principaux de recherche pour la définition des ordinateurs de cinquième génération. On pourra à ce sujet consulter certains articles prospectifs, tels

- **Fifth Generation Kernel Langage** T. Chikayama et al., proc. of Logic Programming Conference 1983.

La machine proposée concerne ce dernier type de langage. Elle n'est pas un interpréteur complet d'un langage de programmation logique, mais plutôt une mémoire spécialisée, gouvernable par des commandes. Elle est adaptée, par sa structure et les services qu'elle offre, à la sorte de conservation d'information que nécessite un tel interpréteur. Elle s'acquitte de façon transparente des tâches délicates d'allocation et de récupération des ressources de mémoire, en prenant en compte l'indéterminisme, ce qui constitue une originalité par rapport à une machine LISP. Le traitement de ces problèmes à un niveau proche du matériel permet d'espérer de bonnes performances pour les mécanismes utilisateurs.

La machine sera le coeur d'un système au sein duquel elle sera commandée par un mécanisme interpréteur d'un langage apparenté à PROLOG, et consistant typiquement en l'exécution de micro-programmes par une unité spécialisée. Des mémoires seront ajoutées pour contenir les micro-programmes et diverses informations temporaires utilisées pour le fonctionnement de l'interpréteur. La machine, quant à elle, sera utilisée pour représenter, sous la forme d'arborescences binaires appelées "termes" et contenant des "variables" aussi bien la sorte d'état correspondant à la sémantique du langage logique offert que la forme interne des programmes rédigés en ce langage.
Dans ce document, on ne donne pas la description d'un interpréteur particulier. Il est simplement évoqué comme étant l'"utilisateur" qui émet des commandes à la machine.

L'état de la machine peut être décomposé en un "état de désignation de termes" vif et un "état de sauvegarde". Leur évolution dépend de deux groupes de commandes distincts.
L'état vif évolue par extension des termes mémorisés, selon les principes de construction et de substitution de termes aux variables. Les accès sont réglés par un système de nommage dynamique. L'utilisateur peut consulter les termes mémorisés. La seule opération de "calcul" offerte est la comparaison, sous une forme restreinte.
Par la commande "sauvegarder", l'utilisateur peut demander à la machine de prendre un cliché de l'état vif à cet instant et de l'ajouter à l'état de sauvegarde. Il peut continuer à faire évoluer l'état vif. Par la commande "reprendre", il peut réinstaller comme état vif le dernier état sauvegardé. Les effets des commandes passées entre la sauvegarde et la reprise, et qui ont affecté l'état vif, sont annulés; il s'agit d'une sorte de retour dans le passé. Ce mécanisme sera la base de l'interprétation de l'indéterminisme dans le langage logique.

La présentation comprend deux parties.
La partie "spécification" pose une relation entre les informations soumises à la mémoire et les informations qu'elle rend. L'effet mémorisateur est rendu par un état abstrait, d'ordre mathématique, sans rapport direct avec les rouages internes de la réalisation. Cette spécification est destinée à être comprise par les concepteurs de mécanismes utilisateurs, et à servir de base pour prouver que ces mécanismes sont corrects.
La partie "réalisation" expose quels composants élémentaires sont mis en oeuvre et comment ils sont assemblés pour constituer une machine qui, aux bornes, c'est-à-dire dans ses échanges d'information avec l'utilisateur, se comporte comme la machine abstraite énoncée dans la spécification.

partie 2

SPECIFICATION

NOTION DE BASE: ESPACE DE TERMES

Un triplet ‹TERMES, FEUILLES, CONSTRUCTEURS› est un **espace_de_termes** si

⊢ CONSTRUCTEURS est une partie finie de (TERMES×TERMES) → TERMES

⊢ FEUILLES ⊂ TERMES

⊢ ∀ g1,d1,g2,d2 ∈ TERMES ∀ cons1,cons2 ∈ CONSTRUCTEURS
cons1(g1,d1) = cons2(g2,d2)
⇒
(cons1=cons2 et g1=g2 et d1=d2)

⊢ ∀ t ∈ TERMES
t ∉ FEUILLES
⇐⇒
∃ g,d ∈ TERMES ∃ cons ∈ CONSTRUCTEURS t=cons(g,d)

Commentaire:

Les TERMES sont des arbres binaires. Ils ont été préférés aux termes n-aires pour leur rusticité adaptée au degré de matérialité de la mise-en-oeuvre projetée et pour leur souplesse. Ils ont le même pouvoir expressif que les termes n-aires et il peuvent servir de composants élémentaires pour une grande variété de structures, en particulier les listes de longueur variable. Leur morphologie homogène se prête bien à l'allocation dynamique d'un réservoir de cellules de mémoire toutes identiques et interchangeables.

Il est prévu plusieurs constructeurs, en pratique un petit nombre. Grâce à eux, les applications disposent d'arguments de discrimination basés sur le type des termes construits, plutôt que sur un codage employant des feuilles réservées. On espère ainsi une plus grande efficacité.

Si la définition donnée ici est très générale et tolère en particulier les termes infinis, les contraintes d'utilisation énoncées au sein de la spécification limitent le domaine représentable dans la machine aux termes finis arborescents.

NOTIONS DERIVEES: TERMES_CONSTRUITS, SOUS_TERMES, REMPLACEMENT

⊢ ∀ t
t ∉ TERMES_CONSTRUITS
<=>
∄ cons ∈ CONSTRUCTEURS ∄ tg,td ∈ TERMES t=cons(tg,td)

⊢ SOUS_TERMES ∈ TERMES → parties de TERMES
⊢ ∀ t ∈ TERMES
(t ∈ FEUILLES et SOUS_TERMES(t) = { })
ou ∄ cons ∈ CONSTRUCTEURS ∄ tg,td ∈ TERMES
t=cons(tg,td)
et SOUS_TERMES(t) =
{tg} ∪ {td} ∪ SOUS_TERMES(tg) ∪ SOUS_TERMES(td)

⊢ REMPLACEMENT de ... par ... dans ...
∈ (TERMES×TERMES×TERMES) → TERMES
⊢ ∀ t1,t2,t ∈ TERMES
t=t1
=>
REMPLACEMENT de t1 par t2 dans t = t2
et
(t≠t1 et t ∈ FEUILLES)
=>
REMPLACEMENT de t1 par t2 dans t = t
et
(t≠t1 et t ∉ FEUILLES)
=>
∄ cons ∈ CONSTRUCTEURS ∄ tg,td ∈ TERMES
t=cons(tg,td)
et REMPLACEMENT de t1 par t2 dans t =
cons(REMPLACEMENT de t1 par t2 dans tg,
REMPLACEMENT de t1 par t2 dans td)

Commentaire:

Les TERMES_CONSTRUITS et les FEUILLES sont des ensembles disjoints et dont la réunion constitue les termes.
SOUS_TERMES donne l'ensemble des sous-termes d'un terme, non compris lui-même.
REMPLACEMENT remplace toute occurence d'un terme par un autre terme au sein d'un certain terme.

NOTION DE BASE: ESPACE DE LISTES

Un quintuplet ‹LISTES, ELEMENTS, EN_LISTE, LISTE_VIDE, TAILLE_LISTE› est un **espace_de_listes** si

⊢ EN_LISTE ∈ (ELEMENTS×LISTES)→LISTES
⊢ LISTE_VIDE ∈ LISTES
⊢ ∀ e1,e2 ∈ ELEMENTS ∀ l1,l2 ∈ LISTES
EN_LISTE(e1,l1) = EN_LISTE(e2,l1)
⇒
(e1=e2 et l1=l2)
⊢ ∀ l ∈ LISTES
l ≠ LISTE_VIDE
⇔
∃ e1 ∈ ELEMENTS ∃ l1 ∈ LISTES l = EN_LISTE(e1,l1)
⊢ TAILLE_LISTE ∈ LISTES → **entiers**
⊢ TAILLE_LISTE(LISTE_VIDE) = 0
⊢ ∀ e ∈ ELEMENTS ∀ l ∈ LISTES
TAILLE_LISTE(EN_LISTE(e,l)) = TAILLE_LISTE(l) + 1
⊢ ∀ l ∈ LISTES ∃ n ∈ **entiers** TAILLE_LISTE(l) = n

Commentaire:

Une LISTE est une collection ordonnée d'éléments. La fonction EN_LISTE rajoute un élément en tête. TAILLE_LISTE donne le nombre d'éléments d'une liste.

NOTION DERIVEE: SOUS_LISTES

⊢ SOUS_LISTES ∈ LISTES → parties de LISTES
⊢ ∀ l ∈ LISTES
(l = LISTE_VIDE et SOUS_LISTES(l) = { })
ou ∃ e ∈ ELEMENTS ∃ l1 ∈ LISTES
l = EN_LISTE(e,l1)
et SOUS_LISTES(l) = {l1} ∪ SOUS_LISTES(l1)

Commentaire:

Les SOUS_LISTES d'une liste sont les listes obtenues en enlevant progressivement les éléments de tête.

DOMAINES UTILISES POUR LA DEFINITION DE LA MACHINE

données, données1, données2, noms, marques_de_nature, marques_de_comparaison

⊢ **données1**, **données2** sont des ensembles finis disjoints
⊢ **données** = **données1** ∪ **données2**
⊢ **noms** est un ensemble dénombrable
⊢ **marques_de_nature** = {**c1**, **c2**, **v1**, **v2**, **d1**, **d2**, **niv**}
⊢ **marques_de_comparaison** = {**égal**, **différent**, **non_décidé**}

Commentaire:

Cette rubrique spécifie les informations qui s'échangent entre la machine et l'utilisateur lors des commandes, en plus du code de la commande.
En ce qui concerne les **données**, la compétence de la machine se limite à pouvoir les mémoriser et les comparer. Par contre, l'utilisateur disposera en général de mécanismes leur conférant une signification particulière, par exemple arithmétique ou textuelle. Pour plus de commodité, deux formats sont prévus: les **données1** et les **données2**.
En ce qui concerne les **noms**, ils sont sous la maitrise de la machine: elle décide de l'évolution dynamique de leur signification, et elle seule peut les interpréter réellement. L'utilisateur ne peut rien faire des noms que la machine lui révèle, sinon les mémoriser, et les re-citer tôt ou tard lors d'une commande, ou les oublier.
Les diverses **marques** servent dans le cadre de conventions communes entre la machine et l'utilisateur, pour discriminer un cas parmi plusieurs possibles dans certaines situations.

variables1, variables2, variables, niveaux, atomes, cons1, cons2, constructeurs, termes, sous_termes, remplacement, construits1, construits2

⊢ **niveaux** = **entiers**
⊢ **variables1**, **variables2** sont des ensembles dénombrables disjoints
⊢ **variables** = **variables1** ∪ **variables2**
⊢ **variables**, **données**, **niveaux** sont disjoints
⊢ **atomes** = **données** ∪ **niveaux**
⊢ **constructeurs** = {**cons1**, **cons2**}
⊢ ‹**termes**, **variables** ∪ **atomes**, **constructeurs**› est un **espace_de_termes**
avec **termes_construits**, **sous_termes**, **remplacement**
en tant que notions dérivées
TERMES_CONSTRUITS, SOUS_TERMES, REMPLACEMENT
⊢ ∀ t (t ∈ **construits1** <=> ∃ tg,td ∈ **termes** t=**cons1**(tg,td))
⊢ ∀ t (t ∈ **construits2** <=> ∃ tg,td ∈ **termes** t=**cons2**(tg,td))

Commentaire:

Cette rubrique spécifie la sorte d'information conservée par la machine. Parmi les **termes**, seules les données peuvent être échangées avec l utilisateur. Les autres termes restent confinés dans la machine et ne sont accessibles à l'utilisateur qu'indirectement, en pratiquant les commandes répertoriées et des noms décidés par la machine elle-même.

Les **variables** constituent une sorte particulière de feuilles. Elles servent à introduire une relation supplémentaire entre termes, la relation "être une instance". Un terme est une instance d'un autre terme s'il s'obtient en remplaçant certaines des variables de ce dernier par d'autres termes. Dans la littérature, on emploie également les expressions "t2 subsume t1" ou "t2 shématise t1" en place de "t1 est une instance de t2". Pour faciliter l'interprétation de notions propres à certains langages logiques, et éviter des codages peu performants de la part de l'utilisateur, la machine offre deux classes de variables: les **variables1** et les **variables2**.

Les **niveaux** sont des atomes particuliers qui peuvent être captés et conservés par l'utilisateur et qui lui permettent d'effectuer certains contrôles sur l'état de sauvegarde.

La figure 6 illustre comment se répartissent les diverses classes de termes.

$\underline{\perp}$

$\perp \notin$ **termes**

Commentaire:

$\perp$ est un élément utilisé dans la spécification pour jouer le rôle d'"indéfini".

Définition: **désignations**

désignations est l'ensemble ainsi défini:

D $\in$ **désignations**
<=>
D $\in$ **noms** $\rightarrow$ (**termes** $\cup \{\perp\}$)
et $\forall$ n $\in$ **noms**
$\exists$ cons $\in$ **constructeurs** $\exists$ tg,td $\in$ **termes**
D(n) = cons(tg,td)
=> $\exists$ ng,nd $\in$ **noms** (D(ng)=tg et D(nd)=td)

Commentaire:

Une **désignation** est telle que si un terme est désignable, c'est-à-dire s'il existe un nom qui désigne ce terme, tous ses sous-termes le sont aussi.

Une des composantes de l'état de la machine est une désignation et sert à exprimer quels sont, à tout moment, les termes accessibles à l'utilisateur, et via quels noms. L'utilisateur dispose de commandes pour faire évoluer cet état de désignation.

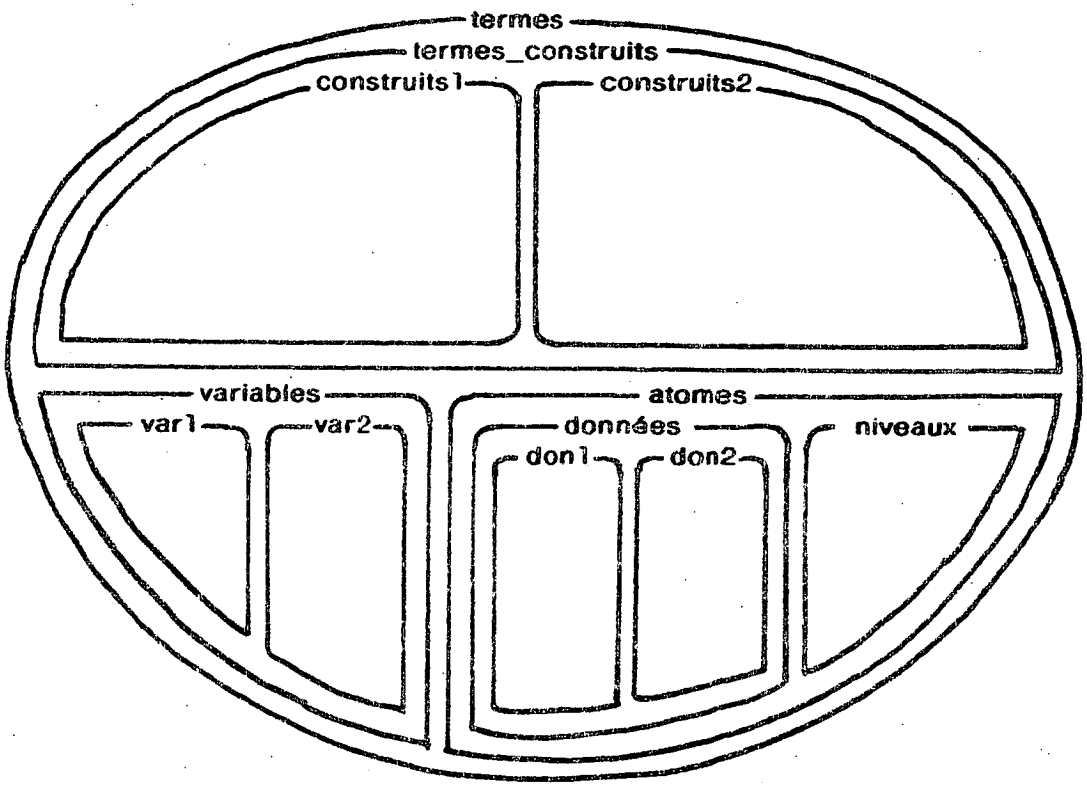


figure 6 - Les **termes**

sauvegardes, sous_sauvegardes, sauve, sauvegarde_vide, taille

⊢ <sauvegardes, termes×termes×termes, sauve, sauvegarde_vide, taille>
est un espace_de_listes avec
sous_sauvegardes en tant que notion dérivée SOUS_LISTES

Commentaire:

Une **sauvegarde** est une pile de triplets de termes. Chaque triplet constitue une strate de la pile. Le nombre de termes dans une strate a été choisi égal à trois pour obtenir une bonne souplesse d'utilisation.
Une des composante de l'état de la machine est une sauvegarde, l'utilisateur dispose de certaines commandes pour modifier cette sauvegade, c'est à dire, empiler une strate, dépiler une strate, capter l'état de sauvegarde courant sous forme d'un niveau et le recycler plus tard en tant que paramètre d'une commande de modification de l'état de sauvegarde à ce moment là.

NATURE DE LA MACHINE: COMMANDES, PARAMETRES, RESULTATS, ETAT.

L'utilisateur soumet des commandes à la machine séquentiellement au cours du temps.

Toute commande a pour composantes un code de commande et en général des composantes complémentaires qui appartiennent à des domaines divers selon la commande, et qui sont précisées sous la rubrique "entrées" dans la spécification de la commande. Le résultat rendu par la machine en réponse à une commande a ses composantes dans des domaines qui dépendent aussi de la commande considérée, et qui sont précisées sous la rubrique "sorties" dans la spécification.

Entre deux soumissions de commandes, la machine a un état qui est un élément de

désignations × sauvegardes.

La soumission d'une commande fait passer l'état de <*Davant*,*Savant*> à <*Daprès*,*Saprès*>. Entre *Davant*, *Savant*, les entrées de la soumission, *Daprès*, *Saprès* et les sorties de la soumission, il y a la relation

CONTRAINTE => EVOLUTION

où CONTRAINTE et EVOLUTION sont les relations posées respectivement sous les rubriques "contrainte" et "évolution" dans la spécification de la commande.

Commentaire:

Les deux composants d'un état <D,S> peuvent se comprendre de la façon suivante.

- D constitue l'"état de désignation des termes". Une famille de commandes le concerne et fait changer la signification des noms.
- S constitue l'"état de sauvegarde". Des commandes spécifiques permettent de le faire évoluer a' la manière d'une pile. Ces commandes ont aussi l'effet annexe d'affecter ou consulter l'état de désignation.

Quand l'utilisateur ne respecte pas la contrainte associée à une commande, l'effet est indéfini. Si l'utilisateur est mal conçu, et risque par exemple de citer lors d'une commande un nom ne signifiant rien, ou signifiant un terme par hasard, le concepteur est dans l'impossibilité de prouver que la relation de la rubrique "contraintes" est toujours vraie.

REPERTOIRE DES COMMANDES

Les commandes peuvent être regroupées selon les familles suivantes:

Manipulation de la désignation des termes:
construire, créer_variable, substituer, accéder_gauche, accéder_droite
créer_donnée1, créer_donnée2, accéder_donnée1, accéder_donnée2
tester_nature, comparer
Manipulation des sauvegardes:
sauvegarder, reprendre, obtenir_niveau_courant, couper
Récupération de mémoire:
réduire
Initialisation:
initialiser

Commande construire

Entrées: ng,nd ∈ **noms** m ∈ {**c1**,**c2**}
Sorties: nt ∈ **noms**
Contrainte: $D_{avant}(ng) \neq \perp$ et $D_{avant}(nd) \neq \perp$
Evolution:
(m=**c1** et $D_{après}(nt)$=**cons1**($D_{avant}(ng)$,$D_{avant}(nd)$))
ou (m=**c2** et $D_{après}(nt)$=**cons2**($D_{avant}(ng)$,$D_{avant}(nd)$))
et $\forall$ n ∈ **noms** ($D_{avant}(n) \neq \perp$ => $D_{après}(n)=D_{avant}(n)$)
et $S_{après}=S_{avant}$

Commentaire:

La commande **construire** crée la représentation d'un terme construit, accessible ensuite par le nom rendu en résultat.

Commande créer_variable

Entrées: m ∈ {v1,v2}
Sorties: nv ∈ **noms**
Evolution:
(m=**v1** et $D_{après}$(nv) ∈ **variables1**)
ou (m=**v2** et $D_{après}$(nv) ∈ **variables2**)
et D_{avant}(nv)=⊥
et ∀ n ∈ **noms**
D_{avant}(n)≠⊥ => ($D_{après}$(n)=D_{avant}(n) et $D_{après}$(nv)≠D_{avant}(n))
et $S_{après}$=S_{avant}

Commentaire:

La commande **créer_variable** crée une nouvelle variable, accessible ensuite par le nom rendu en résultat. La variable est nouvelle en ce sens qu'elle n'était pas accessible auparavant par désignation.

Commande substituer

Entrées: nv,nt ∈ **noms**
Contrainte:
D_{avant}(nv) ∈ **variables** et D_{avant}(nt)≠⊥
et D_{avant}(nv) ∉ **sous_termes**(D_{avant}(nt))
Evolution:
∀ n ∈ **noms**
D_{avant}(n)≠⊥
=>
$D_{après}$(n) = **remplacement** de D_{avant}(nv) par D_{avant}(nt) dans D_{avant}(n)
et $S_{après}$=S_{avant}

Commentaire:

La commande **substituer** modifie tous les termes accessibles par la désignation courante, en y remplaçant toute occurence d'une variable donnée par un terme donné. Par contre les termes sauvegardés dans la pile ne sont pas affectés.

La machine ne permet de représenter que des termes finis non bouclés, c'est à dire qui ne sont pas sous-termes d'eux-même. L'utilisateur est responsable du respect de cette limitation, la machine ne faisant pas de vérifications pour des raisons d'efficacité.

Commande **créer_donnée1**

Entrées: d ∈ **données1**
Sorties: nd ∈ **noms**
Evolution:
$D_{après}(nd)=d$
et $\forall$ n ∈ **noms** ($D_{avant}(n)\neq\perp$ => $D_{après}(n)=D_{avant}(n)$)
et $S_{après}=S_{avant}$

Commande **créer_donnée2**

Entrées: d ∈ **données2**
Sorties: nd ∈ **noms**
Evolution:
$D_{après}(nd)=d$
et $\forall$ n ∈ **noms** ($D_{avant}(n)\neq\perp$ => $D_{après}(n)=D_{avant}(n)$)
et $S_{après}=S_{avant}$

Commentaire:

Les commandes **créer_donnée1** et **créer_donnée2** permettent d'obtenir une désignation pour une donnée de type 1 ou 2. L'utilisateur peut alors la faire intervenir dans des constructions de termes grâce aux commandes telles que **construire** ou **substituer**.

Commande **accéder_donnée1**

Entrées: nd ∈ **noms**
Sorties: d ∈ **données1**
Contrainte: $D_{avant}(nd)$ ∈ **données1**
Evolution:
$d=D_{avant}(nd)$ et $D_{après}=D_{avant}$
et $S_{après}=S_{avant}$

Commande **accéder_donnée2**

Entrées: nd ∈ **noms**
Sorties: d ∈ **données2**
Contrainte: $D_{avant}(nd)$ ∈ **données2**
Evolution:
$d=D_{avant}(nd)$ et $D_{après}=D_{avant}$
et $S_{après}=S_{avant}$

Commentaire:

Les commandes **accéder_donnée1** et **accéder_donnée2** restituent la donnée de type 1 ou 2 désignée.

Commande accéder_gauche

Entrées: n ∈ **noms**
Sorties: ng ∈ **noms**
Contrainte: $D_{avant}(n)$ ∈ **termes_construits**
Evolution:
∃ cons ∈ **constructeurs** ∃ td ∈ **termes** $D_{avant}(n)=cons(D_{avant}(ng),td)$
et $D_{après}=D_{avant}$
et $S_{après}=S_{avant}$

Commande accéder_droite

Entrées: n ∈ **noms**
Sorties: nd ∈ **noms**
Contrainte: $D_{avant}(n)$ ∈ **termes_construits**
Evolution:
∃ cons ∈ **constructeurs** ∃ tg ∈ **termes** $D_{avant}(n)=cons(tg,D_{avant}(nd))$
et $D_{après}=D_{avant}$
et $S_{après}=S_{avant}$

Commentaire:

Les commandes **accéder_gauche** et **accéder_droite** permettent d'accéder aux sous-termes des termes couramment accessibles.

Commande tester_nature

Entrées: n ∈ **noms**
Sorties: m ∈ **marques_de_nature**
Contrainte: $D_{avant}(n) \neq \perp$
Evolution:
($D_{avant}(n)$ ∈ **construits1** et m=**c1**)
ou ($D_{avant}(n)$ ∈ **construits2** et m=**c2**)
ou ($D_{avant}(n)$ ∈ **variables1** et m=**v1**)
ou ($D_{avant}(n)$ ∈ **variables2** et m=**v2**)
ou ($D_{avant}(n)$ ∈ **données1** et m=**d1**)
ou ($D_{avant}(n)$ ∈ **données2** et m=**d2**)
ou ($D_{avant}(n)$ ∈ **niveaux** et m=**niv**)
et $D_{après}=D_{avant}$
et $S_{après}=S_{avant}$

Commentaire:

La commande **tester_nature** révèle la nature du terme désigné parmi les diverses classes ultimes de termes illustrées sur la figure 6.

Commande comparer

Entrées: n1,n2 ∈ **noms**
Sorties: m ∈ **marques_de_comparaison**
Contrainte: D*avant*(n1)≠⊥ et D*avant*(n2)≠⊥
Evolution:
D*après*=D*avant*
et m=**égal** => D*avant*(n1)=D*avant*(n2)
et m=**différent** => D*avant*(n1)≠D*avant*(n2)
et
m=**non_décidé**
=>
(D*avant*(n1) ∈ **termes_construits** et D*avant*(n2) ∈ **termes_construits**)
et S*après*=S*avant*

Commentaire:

La commande **comparer** ne permet pas de comparer de façon systématiquement décisive deux termes construits quelconques. Dans le cas de deux termes construits, cette opération, nécessite généralement un parcours des deux termes qui ferait double emploi, dans les applications visées, avec les parcours commandés par l'utilisateur pour ses propres raisons.
Si deux termes construits sont égaux, la machine répond en général non-décidé, mais il se peut qu'elle réponde égal si la méthode de réalisation fait qu'elle parvient à cette décision de façon directe.

Commande sauvegarder

Entrées: n1,n2,n3 ∈ **noms**
Contrainte: D*avant*(n1)≠⊥ et D*avant*(n2)≠⊥ et D*avant*(n3)≠⊥
Evolution:
D*après*=D*avant*
et S*après* = **sauve**(<D*avant*(n1),D*avant*(n2),D*avant*(n3)>,S*avant*)

Commentaire:

La commande **sauvegarder** permet à l'utilisateur de conserver un vecteur d'état constitué de trois termes, tout en poursuivant le processus de transformation de la collection des termes couramment désignables.
Le vecteur d'état peut être récupéré par la suite tel qu'il a été sauvegardé, en pratiquant la commande **reprendre**. Il n'est pas affecté par les opérations qui se déroulent entre sa sauvegarde et sa reprise; en particulier, les commandes **substituer** n'effectuent pas de remplacements de variables dans les termes sauvegardés. L'ordre des sauvegardes et des reprises se fait selon le principe que le dernier sauvegardé est le premier repris; la mémoire de sauvegarde fonctionne donc en pile.

Commande **reprendre**

Sorties: n1,n2,n3 $\in$ **noms**
Contrainte: $S_{avant} \neq$ **sauvegarde_vide**
Evolution:
S_{avant} = **sauve**($<D_{après}(n1), D_{après}(n2), D_{après}(n3)>$, $S_{après}$)

Commentaire:

La commande **reprendre**, dans le cas où la pile de sauvegarde n'est pas vide, rend à nouveau accessible un vecteur d'état constitué de trois termes. Seuls ces termes et leurs sous-termes deviennent alors couramment désignables. Le processus de transformation en cours avant la commande est donc abandonné et remplacé par la reprise du processus de transformation figuré par le vecteur d'état récupéré.

Commande **obtenir_niveau_courant**

Sorties: np $\in$ **noms**
Evolution:
$D_{après}(np)$=**taille**(S_{avant})
et $\forall$ n1 $\in$ **noms** ($D_{avant}(n1) \neq \perp$ => $D_{après}(n1)=D_{avant}(n1)$)
et $S_{après}=S_{avant}$

Commentaire:

Les commandes **obtenir_niveau_courant** et **couper** sont à la disposition de l'utilisateur pour lui permettre d'effectuer un controle complexe de la gestion des sauvegardes.

La commande **obtenir_niveau_courant** rend la taille de la pile courante, désignable en tant que niveau. Comme pour n'importe quel atome, l'utilisateur peut conserver des niveaux de piles, les comparer et construire des termes les contenant. Cependant, bien que les niveaux soient des entiers, l'utilisateur ne dispose pas de commandes arithmétiques pour les manipuler; la seule commande susceptible de les interpréter est **couper**.

Par propriétés des commandes du répertoire, les niveaux couramment désignables sont toujours inférieurs ou égaux à la taille de la pile de sauvegarde courante. De plus, les niveaux conservés dans les éléments d'une pile de sauvegarde quelconque sont toujours strictement inférieurs à la taille de cette pile.

Commande couper

Entrées: np ∈ **noms**
Contrainte: D*avant*(np) ∈ **niveaux**
Evolution:
∀ n1 ∈ **noms**
D*avant*(n1)≠⊥ => D*après*(n1)=**coupé**(D*avant*(n1),D*avant*(np))
et S*après* ∈ **sous_sauvegardes**(S*avant*)
et **taille**(S*après*)=D*avant*(np)

Commentaire:

La commande **couper** permet de supprimer au sommet de la pile de sauvegarde un certain nombres de vecteurs d'état. La nouvelle pile de sauvegarde est la sous-pile de la pile courante dont la taille est égale au niveau donné en paramètre de la commande.
Afin de conserver la propriété mentionnée, les niveaux qui sont accessibles depuis les termes désignables et qui sont supérieurs à la taille de la nouvelle pile sont diminués et rendus égaux à la taille de la nouvelle pile.

Définition: coupé

coupé ∈ (**termes**×**niveaux**)→**termes**
et ∀ t ∈ **termes** ∀ n ∈ **niveaux**
(t ∉ **niveaux** et t ∉ **termes_construits** et **coupé**(t,n)=t)
ou (t ∈ **niveaux** et t≤n et **coupé**(t,n)=t)
ou (t ∈ **niveaux** et t>n et **coupé**(t,n)=n)
ou ∃ cons ∈ **constructeurs** ∃ tg ,td ∈ **termes**
t=cons(tg,td)
et **coupé**(t,n) = cons(**coupé**(tg,n),**coupé**(td,n))

Commentaire:

coupé(t,n) donne le terme déduit de t en remplaçant dans ce dernier tout niveau supérieur à n par n.

Commande **réduire**

Entrées: n1,n2,n3 ϵ **noms**
Sorties: nn1,nn2,nn3 ϵ **noms**
Contrainte: $D_{avant}(n1) \neq \perp$ et $D_{avant}(n2) \neq \perp$ et $D_{avant}(n3) \neq \perp$
Evolution:
$D_{après}(nn1)=D_{avant}(n1)$ et $D_{après}(nn2)=D_{avant}(n2)$ et $D_{après}(nn3)=D_{avant}(n3)$ et $S_{après}=S_{avant}$

Commentaire:

Après une commande **réduire**, les termes couramment accessibles sont réduits à trois termes et leurs sous-termes.
Cette commande concerne la gestion automatique des ressources de mémoire. Elle se situe donc sur un autre plan que les autres commandes du point de vue strict des possibilités de calcul. Cette commande est rendue nécessaire par la nature, délibérément choisie, de la relation entre la machine et son utilisateur. L'utilisateur est extérieur à la machine, en ce sens qu'il a son propre contrôle et ses propres mémoires inconnus de la machine. Toute information créée pour l'utilisateur peut donc à priori être utilisée dans le futur. La commande **réduire** permet à l'utilisateur de faire connaitre les informations, restreintes à trois termes, qui l'intéressent pour la suite. La machine peut alors en déduire que l'information portée par certaines ressources de mémoire n'est plus utile à la représentation de l'état abstrait, parce que devenue inaccessible à l'utilisateur. Elle peut récupérer ces ressources et les recycler pour une nouvelle utilisation.
La commande **reprendre** provoque également le même mécanisme de récupération: ceci est possible car la désignation est alors réduite aux trois noms rendus en résultat. Cependant, sans la commande **réduire**, les séquences ininterrompues de manipulations de la désignation entraineraient l'occupation d'une quantité croissante de mémoire, conduisant inéluctablement à la saturation du système comme celà se produit dans les interpréteurs PROLOG existants.

Commande **initialiser**

Evolution: $S_{après}$=**sauvegarde_vide**

Commentaire:

La commande **initialiser** fait passer la machine dans un état sain. La pile de sauvegarde est vide et aucun terme n'est conservé par la machine.

partie 3

REALISATION

STRUCTURES DE DONNEES ET REPRESENTATION

STRUCTURES DE DONNEES

TYPES DES INFORMATIONS ECHANGEES AVEC L'UTILISATEUR

donnée1
donnée2
marque_de_nature a pour valeurs possibles {**c1,c2,v1,v2,d1,d2,niv**}
marque_de_comparaison a pour valeurs possibles {**égal,différent,non_décidé**}
nom est une structure à plusieurs champs
    indicateur :**marque_de_nature**
    information :**référence** ou **donnée1**

Commentaire:

Les types d'information **donnée1**, **donnée2**, **marque_de_nature**, **marque_de_comparaison** et **nom** correspondent respectivement aux ensembles **données1**, **données2**, **marques_de_nature**, **marques_de_comparaison** et **noms**. Le type **référence** est d'usage purement interne à la machine. Le codage de ces divers types d'information est illustré sur la figure 7.

TYPES D'INFORMATION D'USAGE INTERNE

référence
niveau
{**libre,indécis**}
{**actif,coupé,mort**}

Commentaire:

Le type d'information **référence** permet d'adresser la mémoire des cellules. Le type **niveau** correspond à l'ensemble des **niveaux**. {**libre,indécis**} sont des marques qui servent à indiquer le statut particulier des cellules au format **variable** décrit ci-après. {**actif,coupé,mort**} sont des marques qui servent à indiquer le statut particulier des cellules au format **géniteur** décrit ci-aprés.

FORMATS DE CELLULE

constructeur est un format à plusieurs champs
    gauche :**nom**
    droite :**nom**

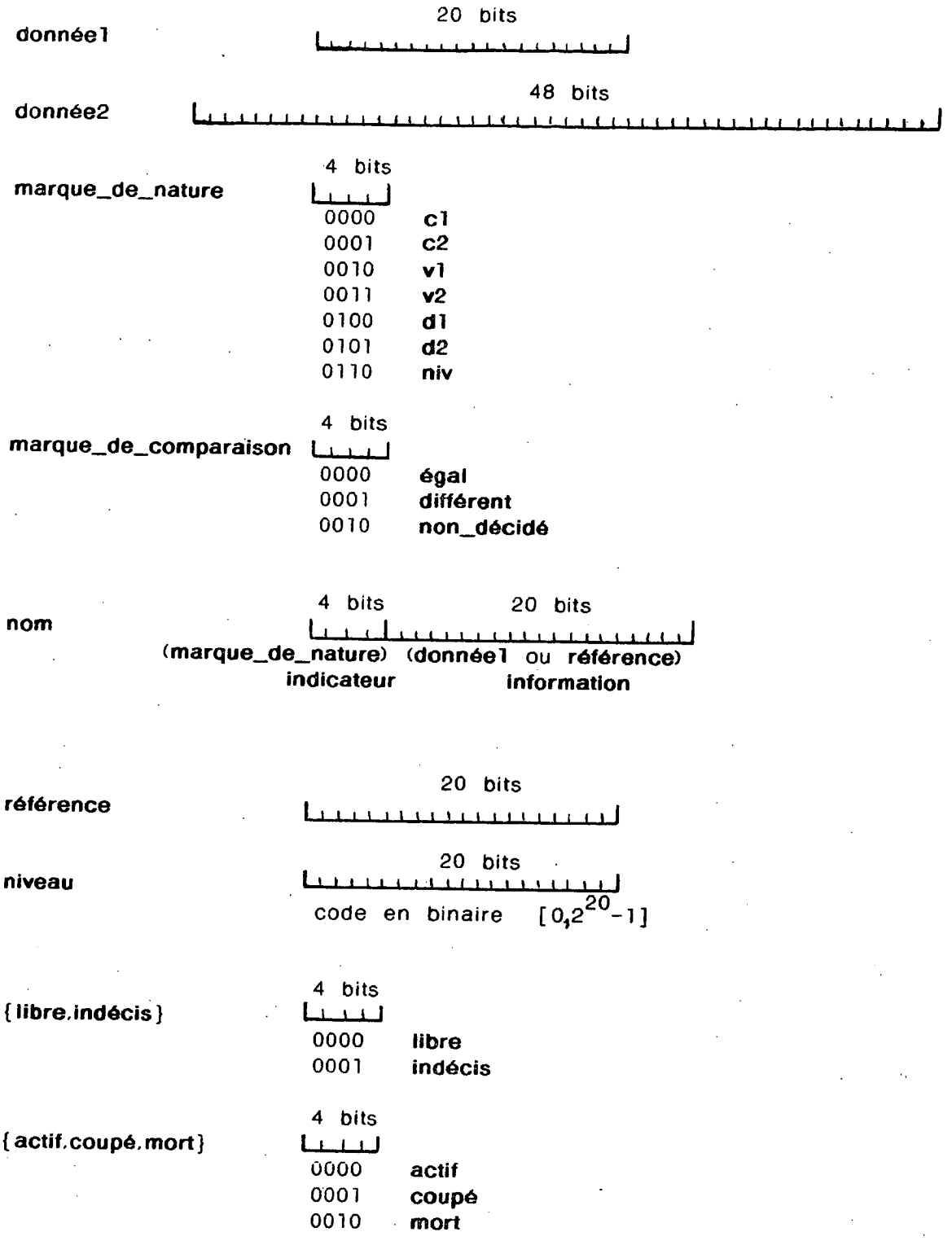


figure 7 - codage des types d'information

variable est un format à plusieurs champs
statut : { **libre**, **indécis** }
géniteur : **référence**
liaison : **nom**

valeur_d2 est un format à un seul champ : **donnée2**

géniteur est un format à plusieurs variantes
discriminées par un champ **nature** : { **actif**, **coupé**, **mort** }
parmi
géniteur_actif si **nature=actif**
qui comporte les autres champs
niveau : **niveau**
précédent : **référence**
n1 : **nom**
n2 : **nom**
n3 : **nom**
géniteur_coupé si **nature=coupé**
qui comporte les autres champs
niveau : **niveau**
précédent : **référence**
géniteur_mort si **nature=mort**

Commentaire:

Une cellule peut être considérée selon quatre formats différents, **constructeur**, **variable**, **valeur_d2** et **géniteur**. La figure 8 illustre la répartition des champs de ces divers formats sur des mots de taille fixe. Les formats **constructeur**, **variable** et **valeur_d2** servent à la représentation des termes. Ces trois formats sont de taille fixe et occupent un mot. Le format **géniteur** sert à la représentation de l'état de sauvegarde. Il regroupe trois formats de tailles différentes, **géniteur_actif**, **géniteur_coupé** ou **géniteur_mort** discernés par le contenu d'un champ commun **nature**. Les formats **géniteur_coupé** et **géniteur_mort** occupent un seul mot. Le format **géniteur_actif** occupe quatre mots chainés comme l'indique la figure 8; ces quatres mots sont considérés comme une seule cellule, référencée par l'adresse du premier mot. L'éclatement de ce format sur quatre mots chainés plutôt que sur des mots consécutifs à été choisi pour que le mécanisme d'allocation de mémoire gère des entités de taille fixe.

MEMOIRE DES CELLULES

La machine comporte une collection de cellules, référencées par une information de type **référence**. Chaque cellule peut être considérée indifféremment sous un quelconque des divers formats pré-cités. Les mécanismes qui sont décrits dans ce document exploitent des fonctions d'accès prédéfinies aux champs des cellules car ces champs ne sont pas accessible toujours isolément. Dans la description des mécanismes, les fonctions d'accès aux cellules sont notées d'une façon qui implique sous quel format la cellule référencée est considérée.

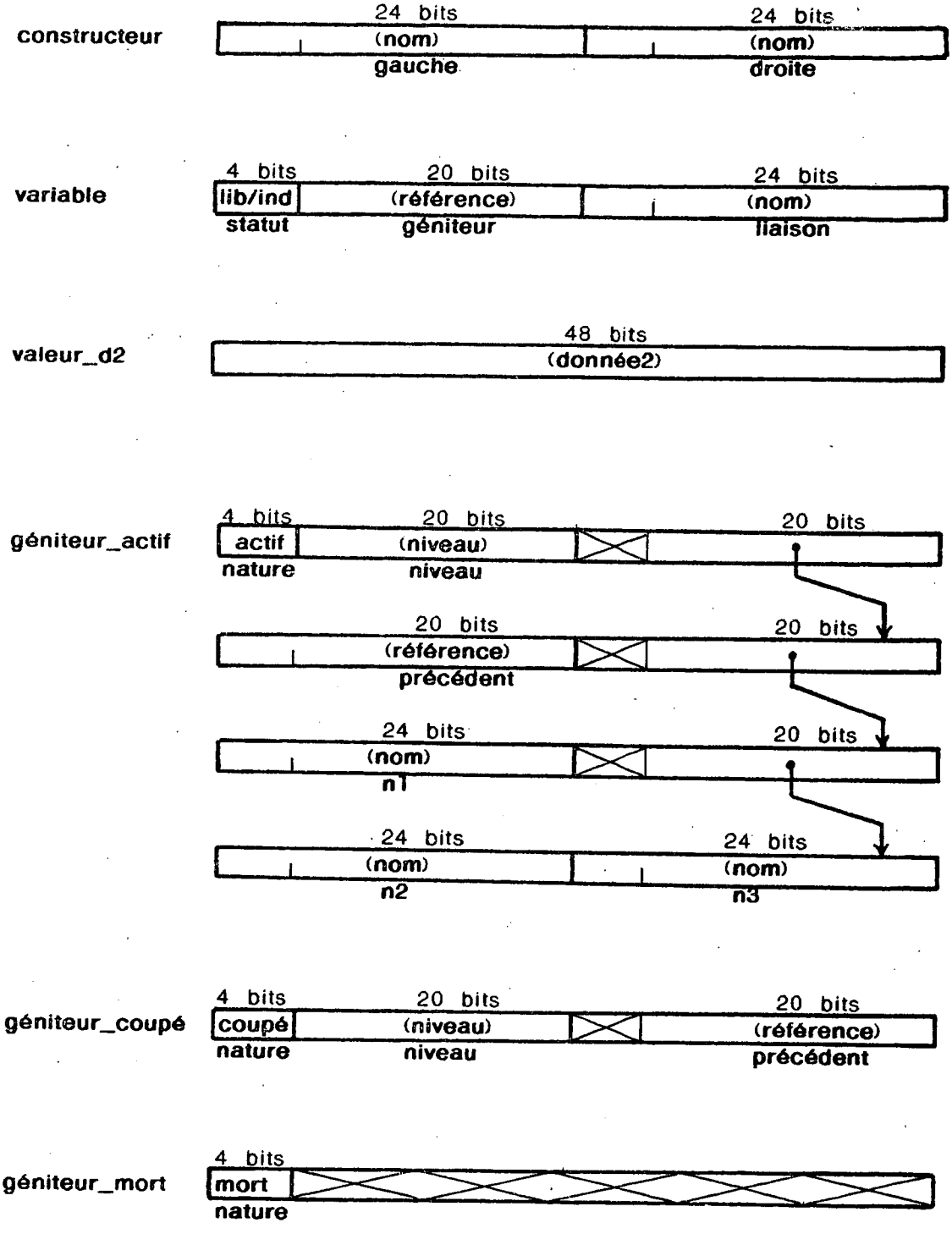


figure 8 - formats de cellules

Ces fonctions d'accès sont les suivantes, *r* étant la référence de la cellule:

gauche[*r*]	champ **gauche**
droite[*r*]	champ **droite**
état_liaison[*r*]	champs **statut.géniteur**
liaison[*r*]	champ **liaison**
valeur_d2[*r*]	**valeur_d2**
état_géniteur[*r*]	champs **nature.niveau**
précedent[*r*]	champ **précédent**
nom_1_2_3[*r*]	champs **n1.n2.n3**

Chaque fonction d'accès peut être exploitée soit en lecture, ce qui est noté:

fonction d'accès → ...

soit en écriture, ce qui est noté:

... → fonction d'accès

Commentaire:

Les cellules sont identiques et accessibles sous n'importe quel format parce que le propre de la machine est précisément d'être un système d'allocation de mémoire pour des besoins divers et changeants. Mais, pour être logiquement corrects, les mécanismes décrits doivent accéder aux cellules sous un format correct.

Pour ne pas entrer dans les détails inintéressants de la gestion des accès aux variantes du format **géniteur**, ces détails sont supposés réalisés par les primitives d'accés **état_géniteur**, **précédent** et **nom_1_2_3**: ces accés testent le champ **nature** pour décider des mesures à prendre pour adresser la mémoire. En particulier dans les accés **état_géniteur** la composante **niveau** est non signifiante si **nature=mort**. L'écriture avec **nature=coupé**, si l'état antérieur est **nature=actif**, provoque automatiquement la recopie du champ **précédent** dans le premier mot, de sorte à assurer la cohérence désirée.

REGISTRE GENITEUR COURANT

géniteur_courant :référence

Commentaire:

Le registre **géniteur_courant** contient la référence d'une cellule considérée sous le format **géniteur**. Il repère la tête de la liste des géniteurs qui représente l'état de sauvegarde courant.

METHODES DE REPRESENTATION

NOTIONS DE BASE

références

⊢ **références** est fini
⊢ **données1** = **références**
⊢ **noms** = **marques_de_nature**×**références**

Définition: **indicateur, information**

indicateur ∈ **noms** → **marques_de_nature** **information** ∈ **noms** → **références**
et ∀ m ∈ **marques_de_nature** ∀ r ∈ **références**
indicateur(<m,r>)=m et **information**(<m,r>)=r

états_de_cellule, gauche, droite, statut, géniteur, liaison, valeur_d2, nature, niveau, précédent, n1, n2, n3

⊢ **états_de_cellule** est fini

⊢ **gauche, droite** ∈ **états_de_cellule** → **noms**
⊢ ∀ g,d ∈ **noms** ∄ c ∈ **état_de_cellule** (**gauche**(c)=g et **droite**(c)=d)

⊢ **statut** ∈ **états_de_cellule** → {**libre,indécis**}
⊢ **géniteur** ∈ **états_de_cellule** → **références**
⊢ **liaison** ∈ **états_de_cellule** → **noms**
⊢ ∀ s ∈ {**libre,indécis**} ∀ g ∈ **références** ∀ l ∈ **noms** ∄ c ∈ **états_de_cellule**
statut(c)=s et **géniteur**(c)=g et **liaison**(c)=l

⊢ **valeur_d2** ∈ **états_de_cellule** → **données2**
⊢ ∀ d ∈ **données2** ∄ c ∈ **états_de_cellule** **valeur_d2**(c)=d

⊢ **nature** ∈ **états_de_cellule** → {**actif,coupé,mort**}
⊢ **niveau** ∈ **états_de_cellule** → **niveaux**
⊢ **précédent** ∈ **états_de_cellule** → **références**
⊢ **n1,n2,n3** ∈ **états_de_cellule** → **noms**
⊢ ∀ i ∈ **niveaux** ∀ p ∈ **références** ∀ n1,n2,n3 ∈ **noms** ∄ c ∈ **états_de_cellule**
nature(c)=**actif** et **niveau**(c)=i et **précedent**(c)=p
et **n1**(c)=n1 et **n2**(c)=n2 et **n3**(c)=n3
⊢ ∀ i ∈ **niveaux** ∀ p ∈ **références** ∄ c ∈ **états_de_cellule**
nature(c)=**coupé** et **niveau**(c)=i et **précédent**(c)=p
⊢ ∄ c ∈ **états_de_cellule** **nature**(c)=**mort**

Définition: **configurations**

configurations = **références** → **états_de_cellule**

Commentaire:

données1 est assimilé à **références.**
indicateur et **information** sont les fonctions de sélection de champ de la structure **nom.**
Les **états_de_cellule** sont les informations brutes que la mémoire est capable de conserver dans une cellule. Les diverses fonctions posées traduisent la capacité des cellules et les sélections de champs des divers formats sous lesquels une cellule peut être considérée.
Les **configurations** sont les états possible de la mémoire des cellules.

REPRESENTATION D'UNE DESIGNATION DE TERMES

Définition: d_variable_liée

n est **d_variable_liée** dans M si
n ∈ **noms** M ∈ **configurations**
et **indicateur**(n)=**v1** ou **indicateur**(n)=**v2**
et **statut**(M(**information**(n)))=**indécis**
et **nature**(M(**géniteur**(M(**information**(n)))))≠**mort**

Définition: d_terminal

n est **d_terminal** dans M si
n ∈ **noms** M ∈ **configurations**
et non(n est **d_variable_liée** dans M)

Définition: d_ultime

nu est **d_ultime** de n dans M si
n,nu ∈ **noms** M ∈ **configurations**
et ∃ Sn ∈ (**entiers** → **noms**) ∃ i ∈ **entiers**
Sn(0)=n
et ∀ j ∈ **entiers**
j<i
=>
Sn(j) est **d_variable_liée** dans M
et Sn(j+1) = **liaison**(M(**information**(Sn(j))))
et Sn(i) est **d_terminal** dans M
et
indicateur(Sn(i))≠**niv** et nu=Sn(i)
ou
indicateur(Sn(i))=**niv** et **indicateur**(nu)=**niv**
et **information**(nu) est **s_ultime** de **information**(Sn(i)) dans M

Définition: **d_représentation**

M est **d_représentation** de D si
M ∈ **configurations** D ⊂ **désignations**
et ∀ n ∈ **noms**
D(n) ≠ ⊥
=>
∃ nu ∈ **noms**
nu est **d_ultime** de n dans M
et **indicateur**(nu)=**c1**
=>
∃ ng,nd ∈ **noms**
ng est **d_ultime** de **gauche**(M(**information**(nu))) dans M
et nd est **d_ultime** de **droite**(M(**information**(nu))) dans M
et D(n) = **cons1**(D(ng),D(nd))
et **indicateur**(nu)=**c2**
=>
∃ ng,nd ∈ **noms**
ng est **d_ultime** de **gauche**(M(**information**(nu))) dans M
et nd est **d_ultime** de **droite**(M(**information**(nu))) dans M
et D(n) = **cons2**(D(ng),D(nd))
et **indicateur**(nu)=**v1**
=>
D(n) ∈ **var1**
et ∀ nn,nnu ∈ **noms**
D(nn)=D(n) et nnu est **d_ultime** de nn dans M
=>
nnu=nu
et **indicateur**(nu)=**v2**
=>
D(n) ∈ **var2**
et ∀ nn,nnu ∈ **noms**
D(nn)=D(n) et nnu est **d_ultime** de nn dans M
=>
nnu=nu
et **indicateur**(nu)=**d1** => D(n)=**information**(nu)
et **indicateur**(nu)=**d2** => D(n)=**valeur_d2**(M(**information**(nu)))
et **indicateur**(nu)=**niv** => D(n)=**niveau**(M(**information**(nu)))

Commentaire:

Une cellule de référence *r* considérée sous le format **constructeur** représente un terme construit. Les champs **gauche** et **droite** de la cellule contiennent les noms de deux termes *tg* et *td*. Un nom de la forme **c1**:*r* ou **c2**:*r* est un nom direct de **cons1**(tg,td) ou **cons2**(tg,td).

Une cellule considérée sous le format **variable** admet deux interprétations différentes selon l'état des champs **statut** et **géniteur**. Une telle cellule est appelée variable libre si son champ **statut** contient **libre** ou si son champ **géniteur** contient la référence d'un géniteur mort Sinon, c'est à dire si son champ **statut** contient **indécis** et si son champ **géniteur** contient la référence d'un géniteur actif ou coupé, elle est appelée variable liée.

Une variable libre de référence *r* représente un élément *v* des **variables**. Un nom de la forme **v1**:*r* ou **v2**:*r* est un nom direct de *v*. De plus cette cellule est l'unique représentante de *v*; elle est citée depuis toutes les représentations des occurences de *v* dans les termes, de sorte que son remplacement par un terme à l'occasion d'une commande **substituer** est simplement réalisé par une modification de cette cellule.

Une variable liée est un renvoi au terme cité dans son champ **liaison**. Le remplacement de *v* par *t* à l'occasion d'une commande **substituer** est réalisé en rangeant un nom de *t* dans le champ liaison de la variable libre qui représente *v* et en la changeant en variable liée. Le rôle des géniteurs dans la définition du statut libre/liée des variables sera explicité à propos de la représentation de la sauvegarde. Une variable libre est initialement créée avec son champ **statut** valant **libre**; lors d'un remplacement, la marque **indécis** est rangée dans son champ **statut** et la référence du géniteur courant du moment, contenue dans le registre **géniteur_courant**, est rangée dans son champ **géniteur**. Le géniteur courant est toujours **actif** ou **coupé**, de sorte que la variable est alors considérée comme liée; elle demeure liée jusqu'à l'exécution d'une commande **reprendre** qui, pour restaurer l'état de désignation des termes du moment de la sauvegarde, peut nécessiter de rétablir un statut libre pour cette variable. Pour celà, lors de l'exécution d'une commande **reprendre**, le géniteur courant du moment est transformé en géniteur mort, de sorte que toutes les variables qui ont été liées depuis la dernière sauvegarde redeviennent libres. Dans ce cas le statut libre d'une variable est codé indirectement par le fait que son champ **géniteur** contient la référence d'un géniteur mort. Les géniteurs jouent donc un rôle similaire à la "trainée" utilisée dans la plupart des interpréteurs PROLOG.

Une cellule de référence *r* considérée sous le format **valeurd2** contient un élément *d* des **données2**. Un nom de la forme **d2**:*r* est un nom direct de *d*.

Une cellule de reférence *r* considérée sous le format **géniteur_actif** contient dans son champ **niveau** un élément *i* des **niveaux**. Un nom de la forme **niv**:*r* est un nom direct de *i*.

La représentation d'un élément *d* des **données1** ne nécessite aucune cellule. Un nom de la forme **d1**:*d* est un nom direct de *d*.

On remarquera que, à l'exception des noms directs des **données1**, tous les noms sont de la forme *m*:*r*, où *r* est la référence d'une cellule et *m* est une marque qui indique le type de cette cellule, c'est à dire sous quel format elle doit être considérée. La méthode choisie consiste donc à noter le type d'une cellule à chaque occurence de sa référence, plutot que de la noter sur la cellule elle-même. Bien que cette méthode soit légèrement plus coûteuse en taille de mémoire, elle a pour avantages que les **données1** n'occupent aucune cellule et que le test de la nature du terme désigné par un nom est plus rapide. Ce dernier critère est prépondérant dans les applications visées.

En plus des noms directs précédemment énumérés, un nom peut à un moment donné désigner indirectement un terme.

Un premier cas concerne les noms de la forme **v1**:*r* ou **v2**:*r* où *r* est la référence d'une variable liée. Un tel nom, qualifié de **d_variable_liée** dans la configuration de mémoire considérée, peut être le départ d'une chaine de noms de longueur finie, le champ **liaison** de la variable liée citée par un nom étant le nom suivant de la chaine. Le dernier nom de la chaine ne cite pas une variable liée et tous les noms de la chaine désignent le même terme que lui.

Un autre cas de noms indirects concerne les noms de la forme **niv:r** où *r* est la référence d'un géniteur coupé. Un tel nom cite une strate autrefois sauvegardée qui a été détruite par une commande **couper**. La chaine de références issue du champ **précédent** de ce géniteur **coupé** se termine par la référence d'un géniteur **actif** qui représente l'élément des **niveaux** actuellement désigné par ce nom.
Ce mécanisme de désignation indirecte est décrit par la relation **d_ultime** qui associe à un nom un nom direct désignant le même terme.

Tous les noms qui participent internement à la représentations des termes ne sont pas nécessairement du domaine connu de l'utilisateur. Le domaine de définition de la **désignation** D comporte essentiellement des noms directs, plus éventuellement une collection de noms indirects qui dépend de l'historique des commandes. Après une commande **réduire**, ou une commande **reprendre**, la **désignation** D offerte à l'utilisateur ne comporte que des noms directs. Ce sont les trois noms rendus en résultat par la commande, ainsi que les noms directs des représentations des sous-termes accessibles par ces trois noms. Cette restriction de la **désignation**, invisible à l'utilisateur, se manifeste lors de l'exécution des commandes d'accès aux termes **accéder_gauche** et **accéder_droite**, qui ne rendent en résultat que des noms directs. Par contre, entre deux exécutions de **réduire**, c'est à dire pendant que l'utilisateur est en phase d'élaboration de résultats intermédiaires, la **désignation** D peut comporter temporairement des noms indirects: ce sont les noms qui citent des variables récemment créées puis liées par une commande **substituer**, ou des strates récemment sauvegardées puis détruites par une commande **couper**. Pendant ces phases l'utilisateur peut en effet conserver de tels noms pour accéder aux termes.

Cette réduction de la désignation est en accord avec la spécification de la machine qui laisse en fait un choix encore plus vaste à la réalisation. Elle est importante pour le mécanisme de récupération de mémoire: l'invisibilité des géniteurs coupés permet leur récupération après avoir remplacé les occurences de leurs références par les références des géniteurs actifs équivalents; l'invisibilité des variables liées permettrait de remplacer certaines occurences de leurs références, mais pas toutes à cause du partage avec la sauvegarde décrit au paragraphe suivant, par les noms directs équivalents. Ceci accélèrerait les accès aux termes en supprimant la traversée de ces variables liées qui ont servi d'intermédiaires de construction et qui ne servent plus désormais. Ceci permettrait également de récupérer certaines de ces variables. Ce mécanisme n'est cependant pas réalisé dans la version de la machine présentée dans ce document.

La relation **d_représentation** relie la composante D de l'état abstrait et l'état concret de la mémoire. Cette relation est encore indéterministe: pour un élément M des **configurations**, de nombreux éléments D des **désignations** la satisfont. Le complément de détermination dépend en effet de l'historique des commandes et ne peut être pris en charge par une relation faisant intervenir seulement l'état de la mémoire des cellules.

La figure 9 est un exemple de **d_représentation**. On remarquera que plusieurs noms différents, bien que directs, peuvent désigner le même terme, par exemple D(*c1*:C1)=D(*c1*:C3). La représentation permet le partage classique des représentations de sous-termes, par exemple la cellule C7 qui participe à la représentation des termes D(c1:C7), D(c1:C6), D(c1:C3) et D(c1:C1). La cellule C9 est une variable liée, les cellules C5 et C8 sont des variables libres qui représentent deux éléments x et y des **variables1**.

REPRESENTATION D'UNE SAUVEGARDE

Définition: **t_variable_liée**

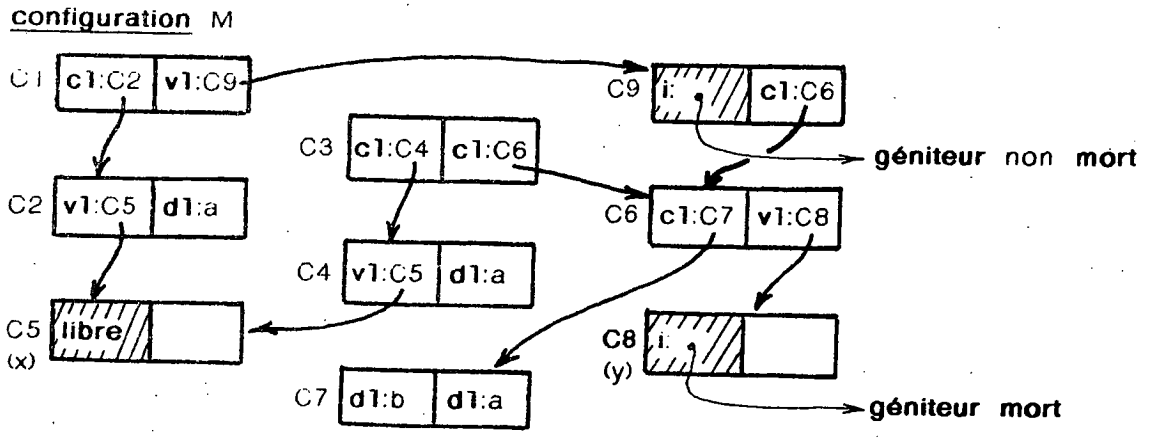
n est **t_variable_liée** pour k dans M si
n ∈ **noms** k ∈ **niveaux** M ∈ **configurations**
et **indicateur**(n)=**v1** ou **indicateur**(n)=**v2**
et **statut**(M(**information**(n)))=**indécis**
et **nature**(M(**géniteur**(M(**information**(n)))) ≠ **mort**
et **niveau**(M(**géniteur**(M(**information**(n)))) ⩽ k

Définition: **t_terminal**

n est **t_terminal** pour k dans M si
n ∈ **noms** k ∈ **niveaux** M ∈ **configurations**
et non (n est **t_variable_liée** pour k dans M)

Définition: **t_ultime**

nu est **t_ultime** de n pour k dans M si
nnu ∈ **noms** k ∈ **niveaux** M ∈ **configurations**
et ∄ Sn ∈ (**entiers** → **noms**) ∄ i ∈ **entiers**
Sn(0)=n
et ∀ j ∈ **entiers**
j<i
=>
Sn(j) est **t_variable_liée** pour k dans M
et Sn(j+1)=**liaison**(M(**information**(Sn(j))))
et Sn(i) est **t_terminal** pour k dans M
et
indicateur(Sn(i))≠**niv** et nu=Sn(i)
ou
indicateur(Sn(i))=**niv** et **indicateur**(nu)=**niv**
et **information**(nu) est **s_ultime** de **information**(Sn(i)) dans M



désignation D

D(c1:C1) = consl(consl(x,a),consl(consl(b,a),y))
D(c1:C2) = consl(x,a)
D(c1:C3) = consl(consl(x,a),consl(consl(b,a),y))
D(v1:C9) = consl(consl(b,a),y)
D(c1:C4) = consl(x,a)
D(c1:C6) = consl(consl(b,a),y)
D(v1:C5) = x
D(c1:C7) = consl(b,a)
D(v1:C8) = y
D(d1:a) = a
D(d1:b) = b

figure 9 - **d_représentation**

Définition: **t_représentation**

M,k est **t_représentation** de D si
M ∈ **configurations** k ∈ **niveaux** D ∈ **désignations**
et ∀ n ∈ **noms**
D(n) ≠ ⊥
=>
∃ nu ∈ **noms**
nu est **t_ultime** de n pour k dans M
et **indicateur**(nu)=**c1**
=>
∃ ng,nd ∈ **noms**
ng est **t_ultime** de **gauche**(M(**information**(nu))) pour k dans M
et nd est **t_ultime** de **droite**(M(**information**(nu))) pour k dans M
et D(n)=**cons1**(D(ng),D(nd))
et **indicateur**(nu)=**c2**
=>
∃ ng,nd ∈ **noms**
ng est **t_ultime** de **gauche**(M(**information**(nu))) pour k dans M
et nd est **t_ultime** de **droite**(M(**information**(nu))) pour k dans M
et D(n)=**cons2**(D(ng),D(nd))
et **indicateur**(nu)=**v1**
=>
D(n) ∈ **var1**
et ∀ nn,nnu ∈ **noms**
D(nn)=D(n) et nnu est **t_ultime** de nn pour k dans M
=>
nnu=nu
et **indicateur**(nu)=**v2**
=>
D(n) ∈ **var2**
et ∀ nn,nnu ∈ **noms**
D(nn)=D(n) et nnu est **t_ultime** de nn pour k dans M
=>
nnu=nu
et **indicateur**(nu)=**d1** => D(n)=**information**(nu)
et **indicateur**(nu)=**d2** => D(n)=**valeur_d2**(M(**information**(nu)))
et **indicateur**(nu)=**niv** => D(n)=**niveau**(M(**information**(nu)))

Définition: **s_ultime**

gu est **s_ultime** de g dans M si
g,gu ∈ **références** M ∈ **configurations**
et ∃ Sr ∈ (**entiers** → **références**) ∃ i ∈ **entiers**
Sr(0)=n
et ∀ j ∈ **entiers**
j<i
=>
nature(M(Sr(j)))=**coupé** et **niveau**(M(Sr(j)))=**niveau**(M(gu))
et Sr(j+1)=**précédent**(M(Sr(j)))
et gu=Sr(i) et **nature**(M(gu))=**actif**

Définition: s_correspondances

s_correspondances = **références** → (**sauvegardes**∪{⊥})

Définition: s_représentation

M est **s_représentation** de SC si
M ∈ **configurations** SC ∈ **s_correspondances**
et ∀ g ∈ **références**
SC(g) ≠ ⊥
=>
∃ gu ∈ **références**
gu est **s_ultime** de g dans M
et
niveau(M(gu))=0
=>
SC(g) = **sauvegarde_vide**
et
niveau(M(gu))≠0
=>
∃ ggu ∈ **références** ∃ D ∈ **désignations**
ggu est **s_ultime** de **précédent**(M(gu)) dans M
et **niveau**(M(ggu)) = **niveau**(M(gu)−1
et M.**niveau**(M(ggu)) est **t_représentation** de D
et SC(g)=**sauve**(<D(**n1**(M(ggu))),D(**n2**(M(ggu))),D(**n3**(M(ggu)))>,SC(ggu))

Commentaire:

La **sauvegarde** S est une liste de triplets de termes. Une telle liste est représentée par des cellules au format **géniteur_actif** ou **géniteur_coupé** chainées par leurs champs **précedent**. Le premier géniteur de la chaine, appelé géniteur courant, est à tout moment repéré par le registre **géniteur_courant**.

Une cellule au format **géniteur_coupé** contient dans son champ **précédent** un renvoi à un autre géniteur. Elle est le vestige d'une strate sauvegardée puis détruite par une commande **couper**. La chaine de références issue de son champ **précédent** est composée d'un nombre fini de références de géniteurs coupés et se termine par la référence d'un géniteur actif qui lui est équivalent. Ce mécanisme de renvoi est décrit par la relation **s_ultime** qui associe à une référence de géniteur la référence du géniteur actif équivalent.

Une cellule au format **géniteur_actif** dont le champ **niveau** contient *0* représente la **sauvegarde_vide**. Par comportement de la machine, cette cellule est unique dans la mémoire; elle est créée à l'initialisation.

Une cellule au format **géniteur_actif** dont le champ **niveau** contient *i* différent de *0* représente une **sauvegarde** de taille *i*. Son champ **précédent** contient la référence d'un géniteur dont l'équivalent actif représente la **sous_sauvegarde** de taille *i*−1 et contient dans ses champs **n1**, **n2** et **n3** des noms qui désignent les trois termes de tête de la **sauvegarde** de taille *i*.
Ces mécanismes sonts décrits par la relation **s_représentation**.

Les termes sauvegardés ne sont pas représentés par recopie, c'est à dire par des ressources de mémoire propres, mais par partage de ressources entre eux et avec les termes de la **désignation** D. En effet, la plupart des termes représentés au niveau *k* sont des instances de termes représentés au niveau *k*−1, c'est à dire n'en diffèrent que par des remplacements de **variables**. Lors d'une commande **sauvegarder** les termes paramètres de la commande sont déjà représentés et l'exécution de la commande consiste à ranger dans les champs **n1**, **n2** et **n3** du géniteur courant des noms directs qui désignent ces termes. Cependant la **désignation** D continue par la suite d'évoluer, notamment par les commandes **substituer** qui ont un effet de bord sur les termes accessibles par la **désignation** D, effet qui ne doit pas perturber les termes sauvegardés. Pour celà les variables liées contiennent dans leur champ **géniteur** la référence d'un géniteur qui indique à partir de quel niveau de sauvegarde cette variable doit être considérée comme liée. En dessous de ce niveau, cette variable, si elle est rencontrée dans la représentation d'un terme sauvegardé, doit être considérée comme libre.

A chaque niveau *k* est associé une **désignation** D*k* qui fait correspondre aux trois noms **n1**, **n2** et **n3** les trois termes sauvegardés à ce niveau. La relation **t_représentation** décrit comment sont reliés l'état de la mémoire des cellules, un niveau *k* et une désignation D*k* propre à ce niveau. Cette relation est analogue à **d_représentation**, la différence étant localisée dans la relation **t_variable_liée** qui prend en compte le niveau pour décider du statut libre/liée des variables.

Le champ **niveau** des géniteurs n'est pas utilisé lors de l'exécution des commandes de manipulation de la **désignation** D. Lors d'une commande **reprendre**, tous les géniteurs équivalents au géniteur courant du moment sont transformés en géniteurs morts. Ceci supprime toutes les occurences d'un niveau et rend ce niveau réutilisable pour une sauvegarde ultérieure. Pour l'exécution des commandes, seul le champ **nature** des géniteurs intervient pour déterminer le statut libre/liée d'une variable. Le champ **niveau** est utilisé par le mécanisme de récupération de mémoire pour connaitre les accès exacts lors du marquage des cellules utiles.

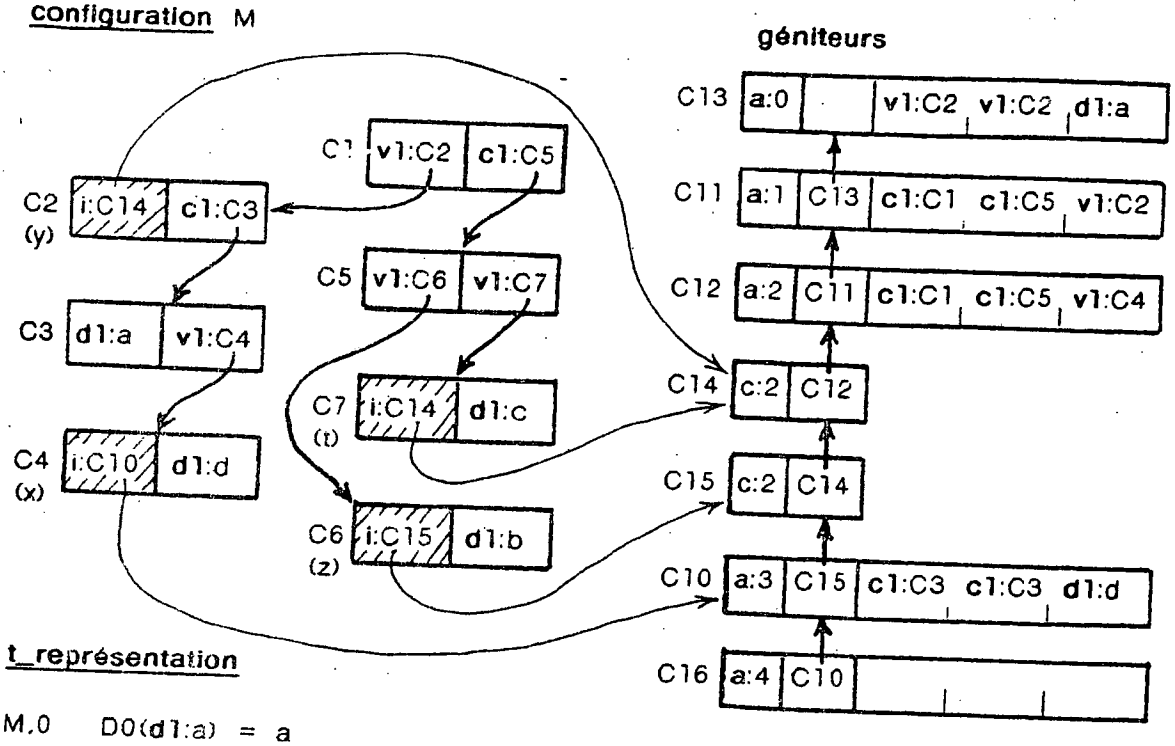
La figure 10 est un exemple de **s_représentation**. La **sauvegarde** représentée par le géniteur C16 est de taille 4. A chaque niveau 0, 1, 2, et 3 correspond une **désignation** D0, D1, D2 et D3. Ces désignations font correspondre des termes aux noms inscrits dans les champs **n1**, **n2** et **n3** des géniteurs actifs du niveau correspondant. C15 et C14 sont des géniteurs coupés équivalents au géniteur actif C12. On remarquera le partage dans les représentations des termes sauvegardés à divers niveaux: par exemple les représentations de D1(**c1**:C1) et D2(**c1**:C1) ont en commun les cellules C1, C2, C5, C6 et C7. Les variables C2, C6 et C7 sont considérées comme liées dans le niveau 2 alors qu'elles sont considérées comme libres dans le niveau 1.

REPRESENTATION DE L'ETAT ABSTRAIT

Définition: **rôles**

rôles = **références** → {**c1**,**c2**,**v1**,**v2**,**d2**,**niv**,**disp**}

configuration M



t_représentation

M,0 D0(d1:a) = a
D0(v1:C2) = y

M,1 D1(c1:C1) = cons1(y,cons1(z,t))
D1(v1:C2) = y
D1(c1:C5) = cons1(z,t)
D1(v1:C6) = z
D1(v1:C7) = t

M,2 D2(c1:C1) = cons1(cons1(a,x),cons1(b,c))
D2(c1:C3) = cons1(a,x)
D2(d1:a) = a
D2(v1:C4) = x
D2(c1:C5) = cons1(b,c)
D2(d1:b) = b
D2(d1:c) = c

M,3 D3(c1:C3) = cons1(a,d)
D3(d1:a) = a
D3(d1:d) = d

s_correspondance SC

SC(C13) = sauvegarde_vide
SC(C11) = sauve(<D0(v1:C2),D0(v1:C2),D0(d1:a)>, SC(C13))
SC(C12) = sauve(<D1(c1:C1),D1(c1:C5),D1(v1:C2)>, SC(C11))
SC(C14) =
SC(C15) =
SC(C10) = sauve(<D2(c1:C1),D2(c1:C5),D2(V1:C4)>, SC(C12))
SC(C16) = sauve(<D3(c1:C3),D3(c1:C3),D3(d1:d)>, SC(C10))

figure 10 - S_représentation

Définition: **nom_cohérent**

n est **nom_cohérent** avec R si
n ∈ **noms**, R ∈ **rôles**
et (**indicateur**(n)=**d1** ou **indicateur**(n)=R(**information**(n)))

Définition: **configuration_cohérente**

M est **configuration_cohérente** avec R si
M ∈ **configurations**, R ∈ **roles**
et
∀ r ∈ **références**
R(r)=**c1** ou R(r)=**c2**
=>
gauche(M(r)) est **nom_cohérent** avec R
et **droite**(M(r)) est **nom_cohérent** avec R
et (R(r)=**v1** ou R(r)=**v2**) et **statut**(M(r))=**indécis**
=>
R(**géniteur**(M(r))) = **niv**
et
nature(M(**géniteur**(M(r)))) ≠ **mort**
=>
liaison(M(r)) est **nom_cohérent** avec R
et R(r)=**niv** et **nature**(M(r))=**coupé**
=>
R(**précédent**(M(r)))=**niv**
et R(r)=**niv** et **nature**(M(r))=**actif** et **niveau**(M(r))≠0
=>
R(**précédent**(M(r))) = **niv**
et ∄ ru ∈ **références**
ru est **s_ultime** de **précédent**(M(r)) dans M
et **n1**(M(ru)) est **nom_cohérent** avec R
et **n2**(M(ru)) est **nom_cohérent** avec R
et **n3**(M(ru)) est **nom_cohérent** avec R

Définition: **représentation**

M,R,g est **représentation** de D,S si
M ∈ **configurations** R ∈ **roles** g ∈ **références** D ∈ **désignations** S ∈ **sauvegardes**
et M est **configuration_cohérente** avec R
et ∀ n ∈ **noms** (D(n)≠⊥ => n est **nom_cohérent** avec R)
et M est **d_représentation** de D
et R(g)=**niv** et **nature**(M(g))≠**mort**
et ∀ r ∈ **références**
R(r)=**niv** et **nature**(M(r))≠**mort**
=>
∄ S ∈ (**entiers** → **références**) ∄ i ∈ **entiers**
S(0)=g
et ∀ j ∈ **entiers** (j<i => S(j+1)=**précédent**(M(S(j))))
et S(i)=r
et ∄ SC ∈ **s_correspondances**
M est **s_représentation** de SC et SC(g)=S
et M,**niveau**(M(g)) est **t_représentation** de D

Commentaire:

Un élément des **rôles** est un statut d'allocation de la mémoire des cellules. La marque **disp** qualifie les cellules disponibles. Les autres marques qualifient les cellules occupées et indiquent le rôle qu'elles jouent.
Un nom qui cite une cellule est **nom_cohérent** si cette cellule est occupée et si son rôle est en accord avec ce nom.
Un état de mémoire est qualifié de **configuration_cohérente** si toutes les références rencontrées dans les cellules occupées sont des références de cellules occupées et dont le rôle est en accord avec le role induit par l'endroit qui fait référence.

La relation **représentation** relie l'état concret constitué de l'état de la mémoire des cellules et du registre **géniteur_courant**, un état d'allocation et l'état abstrait constitué de la **désignation** D et de la **sauvegarde** S. L'état d'allocation concret résulte du mécanisme d'allocation et de récupération de mémoire et dépend de l'état de ses mémoires propres: il est figuré ici par un **role** R.

La mémoire est internement cohérente. Les noms confiés à l'utilisateur au titre de la **désignation** D sont cohérents. Il n'y a qu'une seule **sauvegarde** S, et ses **sous_sauvegardes**, dans le système et tous les géniteurs actifs ou coupés participent à sa représentation: tous ces géniteurs sont accessibles par la chaine de références issue du registre **géniteur_courant**.
De plus, la **désignation** D, qui est en relation de **d_représentation** avec l'état de la mémoire, est également en relation de **t_representation** avec le champ **niveau** du géniteur courant. En effet, la référence du géniteur courant est systématiquement inscrite dans le champ **géniteur** des variables lors de l'exécution des commandes **substituer**. Ainsi la représentation d'un terme quelconque accéssible par la **désignation** D est déjà sous une forme prête à être sauvegardée.

LES MECANISMES DU PROCESSUS D'EXECUTION

```
mécanisme recherche_représentant( nn :nom )     :nom
nn → n
boucle tantque   n.indicateur ∈ {v1,v2}
|   m
|   |   état_liaison[n.information] → el
|   |   si el.statut = indécis alors état_géniteur[n.information] → eg
|   si el.statut = libre alors sortie
|   si eg.nature = mort alors sortie
|   m
|   |   liaison[n.information] → n
|   |   si n.indicateur = niv alors
|   |   |   recherche_gén_représentant(n.information) → n.information

n → résultat
```

Si *nn* fait référence à une variable liée, la valeur de liaison est inspectée, jusqu'à rencontre d'autre chose qu'une variable liée. De plus, si le nom alors obtenu est celui d'un niveau, le géniteur actif représentant du géniteur cité par ce nom est recherché. C'est le nom finalement obtenu qu'on appelle représentant de *nn*.

Si la mémoire est internement cohérente et si le nom *nn* est cohérent, le mécanisme se termine et le résultat est **d_ultime** de *nn*.

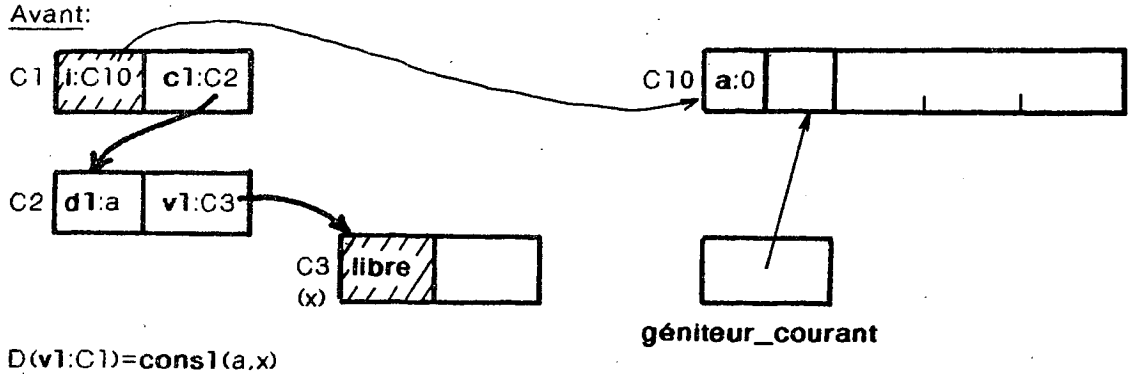
```
mécanisme recherche_gén_représentant( g :référence ) :référence
g → g1
m état_géniteur[g1] → étatgén
boucle tantque étatgén.nature = coupé
|   m précédent[g1] → g1
|   m état_géniteur[g1] → étatgén
g1 → résultat
```

Etant donné une référence à un géniteur, ce mécanisme rend la référence du premier géniteur actif dans la chaine commençant à ce géniteur, et de même niveau que lui.

Si la mémoire est cohérente et si *g* est la référence d'un géniteur non mort, le mécanisme se termine et le résultat est **s_ultime** de *g*.

```
mécanisme construire( ng,nd :nom, mc :{c1,c2} )   :nom
allocation_cellule → ref
mc,ref → résultat
recherche_représentant(ng) → rg
m rg → gauche[ref]
recherche_représentant(nd) → rd
m rd → droite[ref]
```

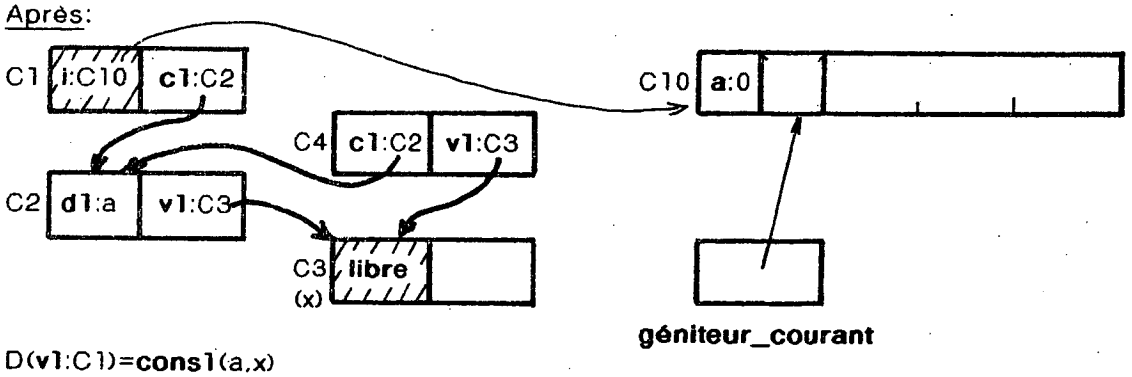
Avant:



D(**v1**:C1)=**cons1**(a,x)
D(**c1**:C2)=**cons1**(a,x)
D(**v1**:C3)=x

Commande: **construire**(**v1**:C1, **v1**:C3, **c2**)

Après:



D(**v1**:C1)=**cons1**(a,x)
D(**c1**:C2)=**cons1**(a,x)
D(**v1**:C3)=x
D(**c2**:C4)=**cons2**(**cons1**(a,x),x)

Résultat: **c2**:C4

figure 11 - **construire**

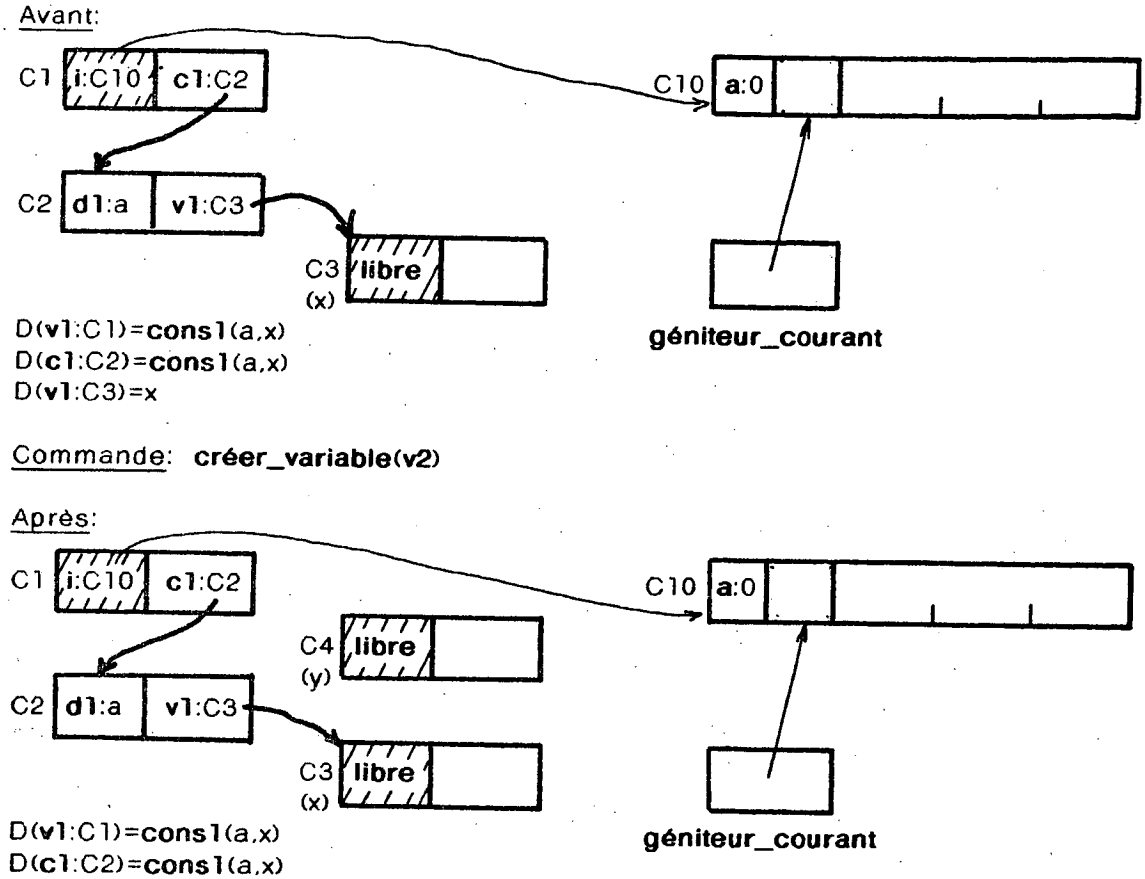


figure 12 - **créer_variable**

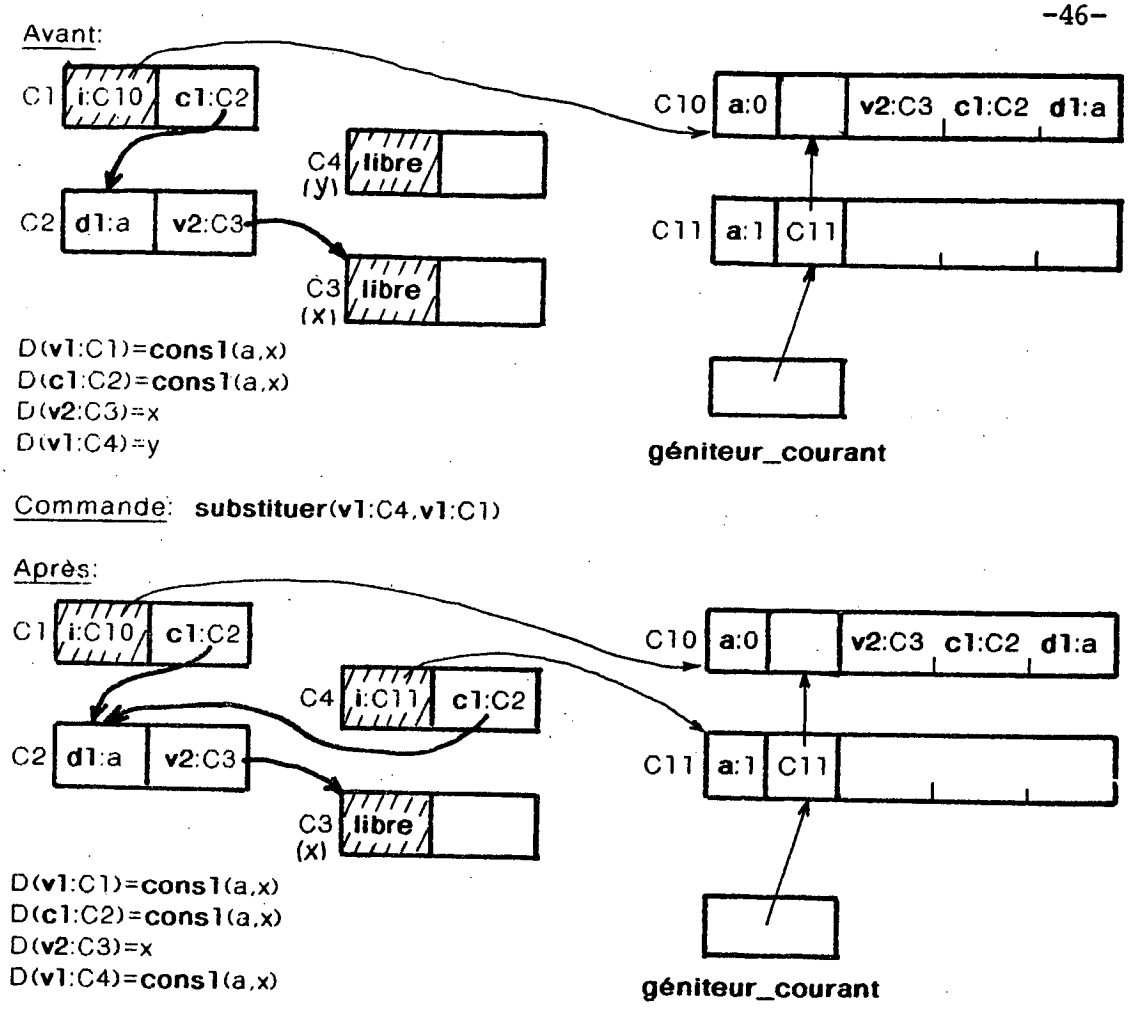


figure 13 - **substituer**

Une cellule est allouée. Elle est considérée sous le format **constructeur**. Deux noms sont rangés dans les champs gauche et droite, qui sont les noms représentants respectifs de *ng* et *nd*. Le résultat consiste en la marque de nature du constructeur et la référence de la cellule. Sur la figure 11, le nom représentant de v1:C1 est c1:C2 parce que la variable en C1 est liée, son statut étant **indécis** et le géniteur correspondant non mort.

Le rôle de la cellule référencée par *ref* passe de **disp** à **c1** (ou **c2**, selon la marque *mc*). La **désignation** D est augmentée du nom **c1**:*ref* (ou **c2**:*ref*).

mécanisme **créer_variable**(mv :{**v1**,**v2**}) :**nom**
allocation_cellule → ref
mv,ref → résultat
m **libre**.□ → **état_liaison**[ref]

Une cellule est allouée. Elle est considérée sous le format **variable**. Son champ **statut** est initialisé à **libre**. Le résultat comporte la marque de nature *mv* et la référence de la cellule. Ceci est illustré sur la figure 12.

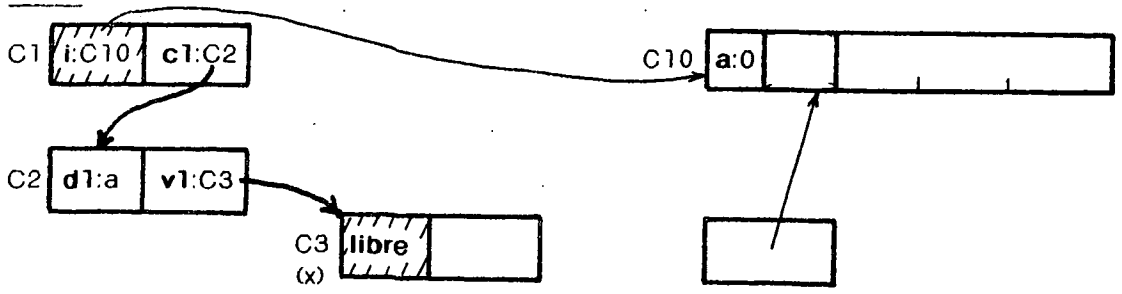
Le rôle de la cellule référencée par *ref* passe de **disp** à **v1** (ou **v2**, selon la marque *mv*). La **désignation** D est augmentée du nom **v1**:*ref* (ou **v2**:*ref*). Cette cellule représente une nouvel élément des **variables** car elle n'est encore référencée depuis nullepart.

mécanisme **substituer**(nv,nt :**nom**)
recherche_représentant(nv) → rv
recherche_représentant(nt) → rt
m
| **indécis**,**géniteur_courant** → **état_liaison**[rv.**information**]
| rt → **liaison**[rv.**information**]

Le nom représentant de *nv* cite une cellule qui, vue sous le format **variable**, est une variable libre. Cette variable subit une liaison, c'est-à-dire que sont rangés dans ses champs **statut**, **géniteur** et **liaison** respectivement la marque **indécis**, la référence du géniteur courant et le nom représentant de *nt*. Sur la figure 13, le paramètre **v1**:C1 contient la référence d'une variable liée; son nom représentant **c1**:C2 est rangé comme valeur de liaison de la variable située en C4.

Le nom *nv* désigne, avant l'exécution de la commande, un élément *v* des **variables**. Le nom *rv*, **d_ultime** de *nv*, est un nom direct de *v*. Il cite donc la variable libre sur laquelle convergent toutes les représentations d'occurences de *v*. Par transformation de cette cellule en variable liée, toutes les occurences de *v* sont remplacées par des occurences du terme désigné par *nt*. Ce remplacement est effectif dans tous les termes accessibles par la **désignation** D. Cependant, le champ **géniteur** de la cellule contient la référence du géniteur courant dont le niveau est supérieur à tous les niveaux couramment sauvegardés. Les **désignations** propres à chaque niveau, en relation de **t_représentation** avec l'état de la mémoire, demeurent inchangées et le remplacement est donc ineffectif dans les termes sauvegardés.

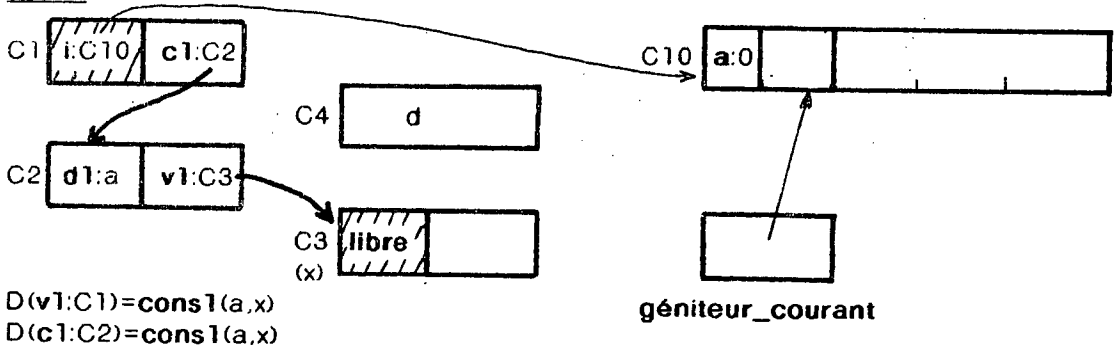
Avant:



D(**v1**:C1)=**cons1**(a,x)
D(**c1**:C2)=**cons1**(a,x)
D(**v1**:C3)=x

Commande: **créer_donnée_2**(d)

Après:



D(**v1**:C1)=**cons1**(a,x)
D(**c1**:C2)=**cons1**(a,x)
D(**v1**:C3)=x
D(**d2**:C4)=d

Résultat: **d2**:C4

figure 14 - **créer_donnée_2**

mécanisme **créer_donnée1**(d :**donnée1**) :**nom**
d1.d → résultat

La création d'une donnée1 consiste simplement à adjoindre la marque de nature à l'information passée en paramètre. Les données1 ne consomment pas de ressource de mémoire.

mécanisme **créer_donnée2**(d :**donnée2**) :**nom**
allocation_cellule → *ref*
d2.ref → résultat
m d → **valeur_d2**[ref]

Une cellule est allouée pour contenir la donnée. Le résultat comporte la référence de cette cellule. Ceci est illustré sur la figure 14.
Le rôle de la cellule référencée par *ref* passe de **disp** à **d2**. La **désignation** D est augmentée du nom **d2**:*ref*.

mécanisme **accéder_donnée1**(nd :**nom**) :**donnée1**
recherche_représentant(nd) → n
n.**information** → résultat

Le résultat est obtenu en retirant au nom représentant de *nd* sa marque de nature.
Le nom *nd* désigne un élement *d* des **données1**. Le nom *n*, **d_ultime** de *nd*, est un nom direct de *d*. Le nom *n* est donc de la forme **d1**:*d*, et son champ **information** rendu en résultat vaut *d*.

mécanisme **accéder_donnée2**(nd :**nom**) :**donnée2**
recherche_représentant(nd) → n
m **valeur_d2**[n.**information**] → résultat

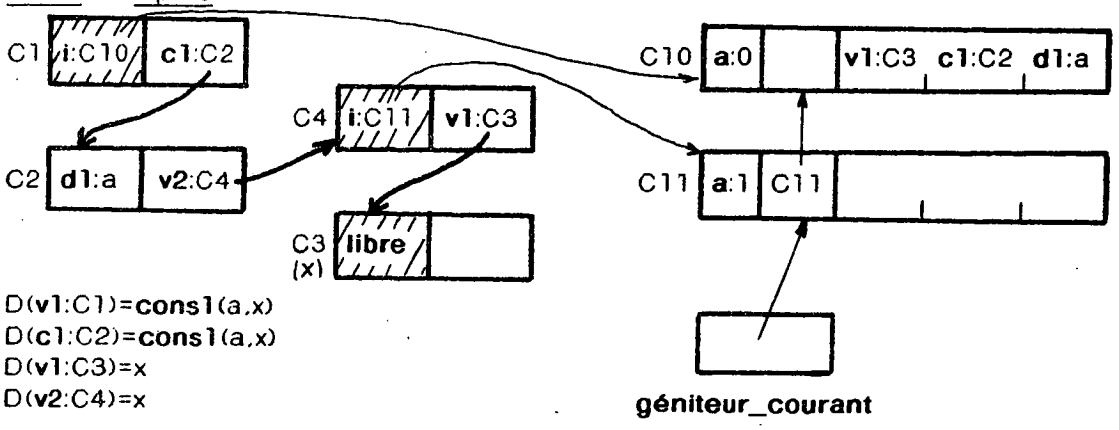
Le contenu de la cellule référencée par le nom représentant de *nd* est rendu en résultat.
Le nom *nd* désigne un élément *d* des **données2**. Le nom *n*, **d_ultime** de *nd*, est un nom direct de *d*. Il cite donc une cellule au format **valeur_d2** dont le contenu, rendu en résultat, vaut *d*.

mécanisme **accéder_gauche**(nc :**nom**) :**nom**
recherche_représentant(nc) → rc
m **gauche**[rc.**information**] → n
recherche_représentant(n) → résultat

Le résultat est le nom représentant du nom trouvé dans le champ **gauche** de la cellule référencée par le nom représentant de *nc*.
Le nom *nc* désigne un terme construit *cons*(*tg*,*td*). Le nom *rc*, **d_ultime** de *nc*, est un nom direct de *cons*(*tg*,*td*). Il cite donc une cellule au format **constructeur**. Le résultat, **d_ultime** du nom *n* contenu dans le champ **gauche** de cette cellule, désigne *tg*. De plus ce nom est un nom direct de *tg*, ce qui permet d'affirmer que aprés une commande **réduire** ou **reprendre** la **désignation** D ne comporte que des noms directs.

Commande: **accéder_droite(v1:C1)**

Avant = Après:



D(**v1**:C1)=**cons1**(a,x)
D(**c1**:C2)=**cons1**(a,x)
D(**v1**:C3)=x
D(**v2**:C4)=x

Résultat: **v1**:C3

figure 15 - **accéder_droite**

```
mécanisme accéder_droite( nc :nom )   :nom
recherche_représentant(nc) → rc
m droite[rc.information] → n
recherche_représentant(n) → résultat
```

Le fonctionnement est identique à celui de la commande **accéder_gauche**. Il est illustré sur la figure 15.

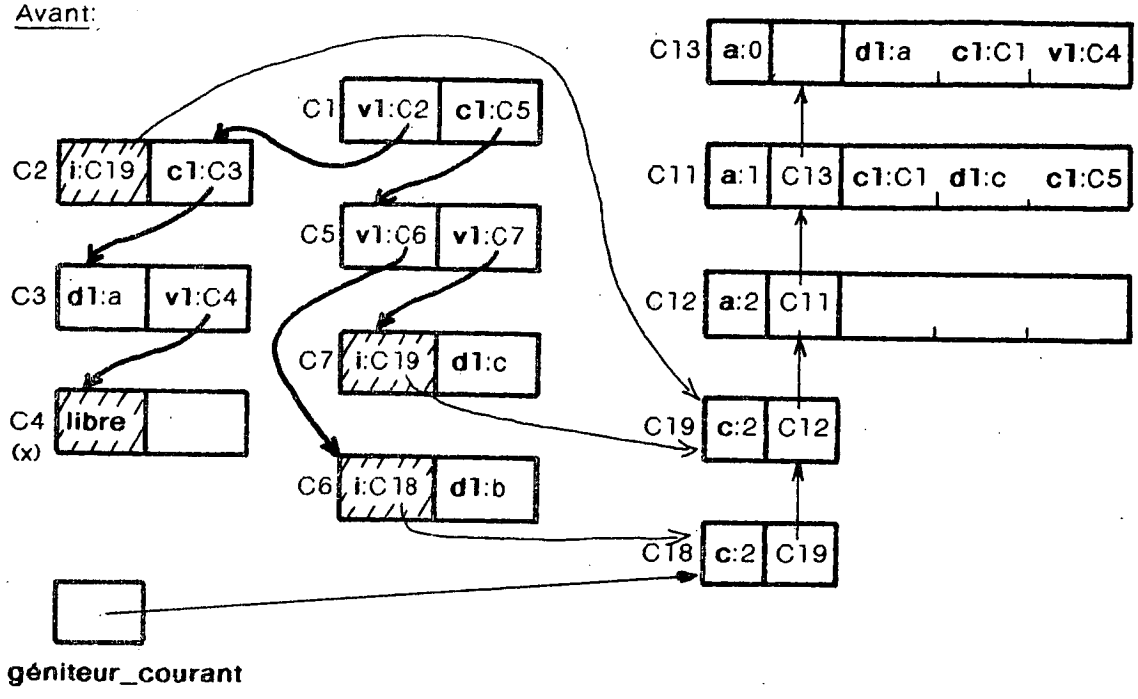
```
mécanisme tester_nature( n :nom )   :marque_de_nature
recherche_représentant(n) → rn
rn.indicateur → résultat
```

Le nom *n* désigne un terme *t*. Le nom *rn*, **d_ultime** de *n*, est un nom direct de *t*. Son champ **indicateur**, rendu en résultat, donne la nature de *t*.

```
mécanisme comparer( n1,n2 :nom )   :marque_de_comparaison
recherche_représentant(n1) → rn1
recherche_représentant(n2) → rn2
si rn1 = rn2  alors égal → résultat
sinonsi rn1.indicateur ≠ rn2.indicateur  alors différent → résultat
sinon
|  cas rn1.indicateur
|  d2 alors
|  |  m valeur_d2[rn1.information] → d1
|  |  m valeur_d2[rn2.information] → d2
|  |  si d1=d2 alors égal → résultat
|  |  sinon différent → résultat
|  niv alors
|  |  m état_géniteur[rn1.information] → eg1
|  |  m état_géniteur[rn2.information] → eg2
|  |  si eg1.niveau = eg2.niveau alors égal → résultat
|  |  sinon différent → résultat
|  c1,c2  alors indécis → résultat
|  v1,v2,d1 alors  différent → résultat
```

Les noms *n1* et *n2* désignent deux termes *t1* et *t2*. Les noms *rn1* et *rn2*, **d_ultimes** de *n1* et *n2*, sont des noms directs de *t1* et *t2*. Si *rn1* est égal à *rn2*, *t1* est égal à *t2* car un nom désigne au plus un terme. Les marques de *rn1* et *rn2* sont significatives de la nature des termes *t1* et *t2*. Si ces marques sont différentes les termes sont donc nécessairement différents. Si les marques sont égales à **d2**, les termes sont des **données2** dont les valeurs sont inscrites dans les cellules au format **valeur_d2** citées par *rn1* et *rn2*. Le résultat est celui de la comparaison de ces valeurs. Si les marques sont égales à **niv**, les termes sont des **niveaux** dont les valeurs sont inscrites dans les champs **niveau** des cellules au format **géniteur** citées par *rn1* et *rn2*. Le résultat est celui de la comparaison de ces valeurs. Si les marques sont égales à **c1** ou **c2**, les termes sont des termes construits dont la comparaison nécessiterait un parcours de cellules couteux et non désiré dans les applications visées. La machine répond **non_décidé** dans ce cas.

Avant:



Commande: **sauvegarder(v1:C2, d1:a, c1:C3)**

Après:

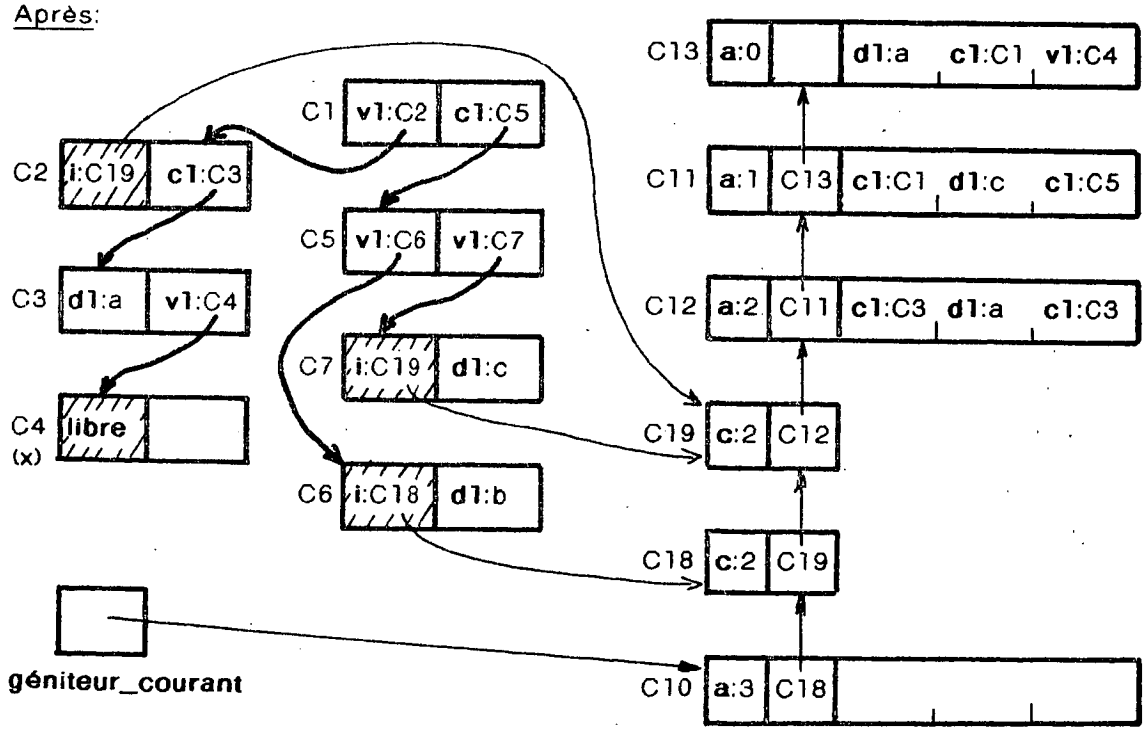


figure 16 - **sauvegarder**

Enfin les seuls cas restants pour les marques sont **v1**, **v2** et **d1**; dans les cas **v1** ou **v2**, *rn1* et *rn2* sont deux noms directs différents de **variables** et les **variables** sont différentes car un élément des **variables** n'est représenté que par une seule cellule. Dans le cas **d1**, *rn1* et *rn2* sont deux noms directs différents de **données1** et ces **données1** sont différents car leurs valeurs sont les champs **information** de *rn1* et *rn2*.

mécanisme **sauvegarder**(n1,n2,n3 :**nom**)
recherche_représentant(n1) → nn1
recherche_représentant(n2) → nn2
recherche_représentant(n3) → nn3
recherche_gen_représentant(**géniteur_courant**) → g1
m nn1,nn2,nn3 → **nom_1_2_3**[g1]
allocation_cell_géniteur → g2
m **état_géniteur**[g1] → eg1
actif.(eg1.**niveau**+1) → eg2
m eg2 → **état_géniteur**[g2]
m **géniteur_courant** → **précédent**[g2]
g2 → **géniteur_courant**

Les noms représentants de n1,n2,n3 sont sauvegardés dans le géniteur actif représentant de **géniteur_courant**. Sur la figure 16, il est en C12. Un nouveau géniteur est créé, son niveau est supérieur d'une unité à l'ancien, c'est lui que repère ensuite **géniteur_courant**.

Avant l'exécution de la commande, le registre **géniteur_courant** contient la référence *gc1* d'un géniteur qui correspond à la **sauvegarde** S_{avant}. Après exécution, le registre **géniteur_courant** contient la référence *g2* d'un géniteur actif. Ce géniteur actif est nouveau, le rôle de *g2* est passé de **disp** à **niv**. La référence *g2* correspond à la nouvelle **sauvegarde** $S_{après}$. En effet, son champ **précédent** contient *gc1*, qui correspond à S_{avant}, et *g1*, qui est **s_ultime** de *gc1*, contient dans ses champs **n1**, **n2** et **n3** trois noms *nn1*, *nn2* et *nn3* qui sont des noms directs des termes à sauvegarder. Ces noms, qui désignent ces termes dans la **désignation** D, les désignent également dans une désignation D_k, ou *k* est le niveau de *gc1*. La désignation D_k est compatible avec l'état de la mémoire selon la relation **t_représentation** grace à l'inscription systèmatique de la référence du géniteur courant dans le champ **géniteur** des variables lors des commandes **substituer**. La nouvelle **sauvegarde** est donc déduite de l'ancienne par ajout en tête du triplet de termes désignés par les paramètres de la commande. Sa taille, *k*+1, est inscrite dans le champ **niveau** du nouveau géniteur courant référencé par *g2*. On remarquera que les noms inscrits dans les géniteurs sont des noms directs, ce qui permet d'affirmer que les désignations D_k propres à chaque niveau ne comportent que des noms directs.

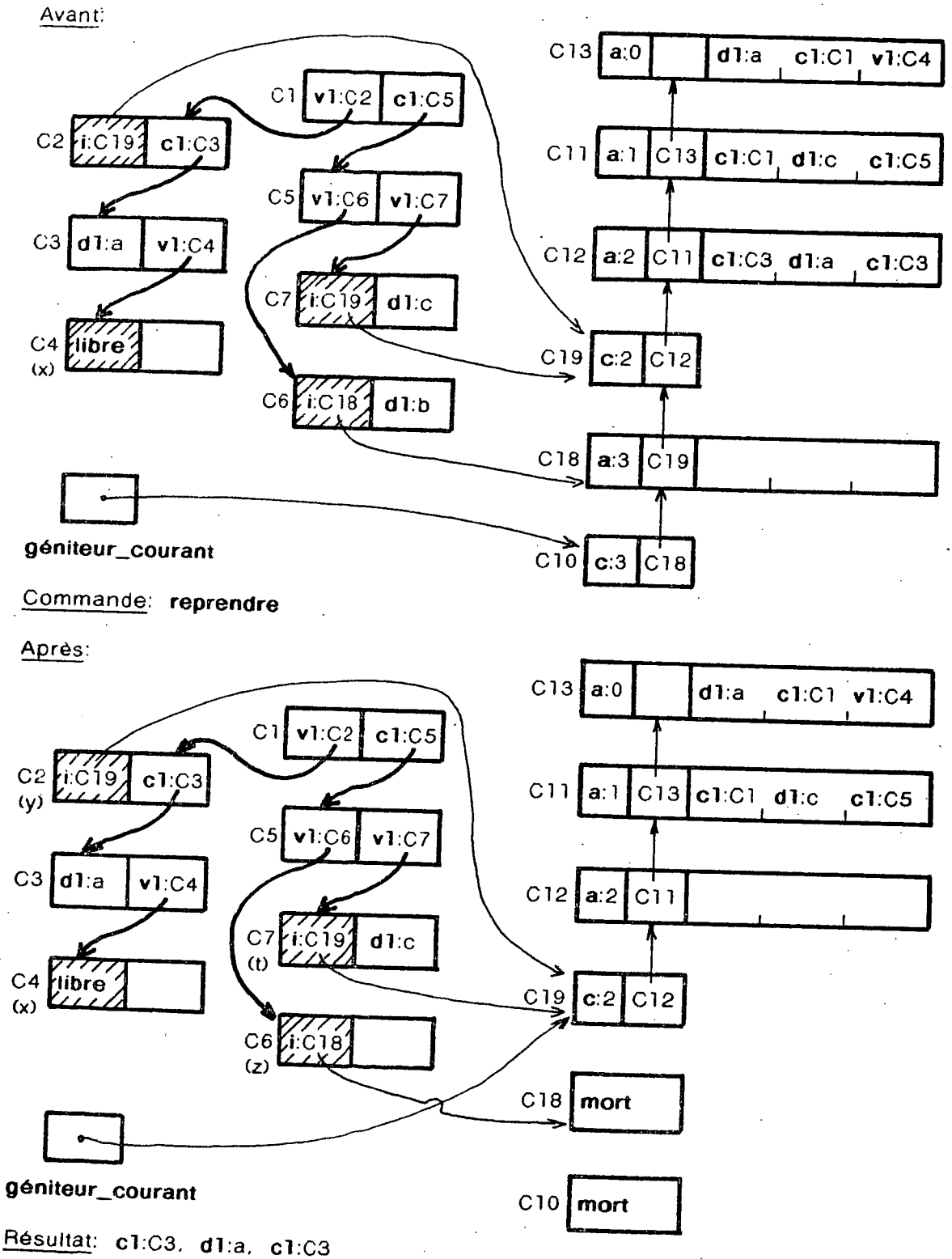


figure 17 - **reprendre**

```
mécanisme reprendre :<nom.nom.nom>
état_géniteur[géniteur_courant] → étatg
1 recalage_sur_reprise(étatg.niveau-1)
2
|   géniteur_courant → g
|   boucle tantque étatg.nature = coupé
|   |   m précédent[g] → ng
|   |   m mort.□ → état_géniteur[g]
|   |   m ng → g
|   |   m état_géniteur[g] → étatg
|   m précédent[g] → géniteur_courant
|   m mort.□ → état_géniteur[g]
recherche_gén_représentant(géniteur_courant) → gr
m nom_1_2_3[gr] → n1,n2,n3
3 démarrage_fournée(géniteur_courant,n1,n2,n3) → géniteur_courant
n1,n2,n3 → résultat
```

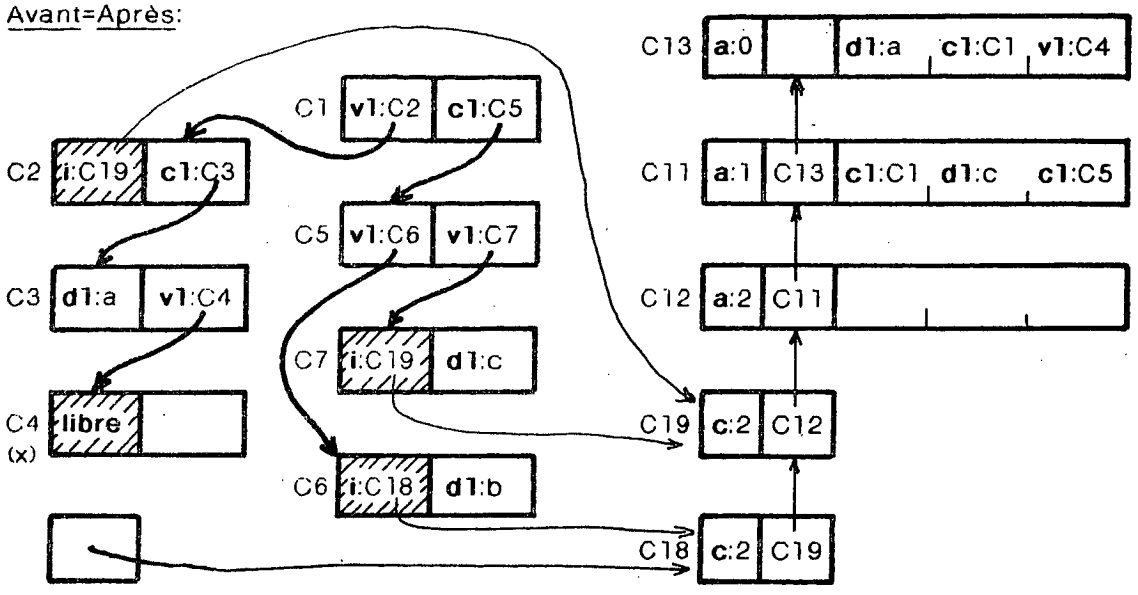
La strate précédente doit être reprise, pour effectuer un retour en arrière et récupérer les trois termes sauvegardés sous forme de trois noms. Sur la figure 17, ces noms sont dans le géniteur situé en C12.

Un recalage est demandé en 1 pour tenir à jour les positions respectives du processus d'exécution et du processus de récupération. Ce dernier, qui traite les strates par niveaux décroissants, risque en effet d'être rattrapé par le processus d'exécution à l'occasion de **reprendre**. Une opération de forçage au niveau inférieur est alors prise à son égard.

En 2, tous les géniteurs de la strate en sommet de pile, c'est-à-dire d'éventuels géniteurs coupés et un actif, situés sur la figure en C10 et C18, sont mis à mort. Ceci a l'effet indirect de rendre à nouveau libres les variables qui avaient été liées pendant la strate qu'on vient d'abandonner. En effet, leur champ **géniteur** pointe sur ces géniteurs désormais morts, ce qui équivaut à être libre. Le processus de récupération s'occupera de son côté de normaliser la situation en notant l'indication de liberté dans la variable elle-même, permettant ainsi, en différé, de détruire les géniteurs morts et de récupérer la place qu'ils occupent.

En 3 il est tiré parti du fait que la désignation est réduite aux trois noms rendus en résultat pour déclencher éventuellement un cycle de récupération de mémoire. Ceci est identique à ce qui est réalisé par une commande **réduire**.

Avant l'exécution de la commande, **géniteur_courant** contient la référence *gc1* d'un géniteur qui correspond à la **sauvegarde** S_{avant} de taille k. S_{avant} étant différente de la **sauvegarde_vide**, le champ **précédent** du géniteur actif *gcr1* équivalent à *gc1* contient la référence *gc2* d'un géniteur qui correspond à la **sous_sauvegarde** de S_{avant} de taille $k-1$. Après exécution, le registre **géniteur_courant** contient *gc2* et la nouvelle **sauvegarde** $S_{après}$ est donc déduite de S_{avant} par retrait de son élément de tête. La nouvelle **désignation** D est la désignation D_{k-1} qui associe aux trois noms rendus en résultat les termes sauvegardés par le niveau $k-1$. Cette désignation $k-1$, compatible avec l'état de la mémoire selon la relation **t_représentation**, est rendue compatible selon la relation **d_représentation** en transformant tous les géniteur de niveau k en géniteurs morts.



Commande: **obtenir_niveau_courant**

Resultat: **niv**:C18 avec D_aprés(**niv**:C18)=2

figure 18 - **obtenir_niveau_courant**

```
mécanisme obtenir_niveau_courant   :nom
niv,géniteur_courant → résultat
```

Le résultat est la référence du géniteur courant, avec l'indicateur **niv**. Ceci est illustré sur la figure 18.
Ce nom désigne en effet la taille de la **sauvegarde** S, directement si le géniteur courant est actif, indirectement s'il est coupé.

```
mécanisme couper( n :nom )
m état_géniteur[n.information] → eg_cible
m état_géniteur[géniteur_courant] → eg
si eg_cible.niveau < eg.niveau alors
|   1 recalage_sur_coupure(eg_cible.niveau)
|   2
|   |   géniteur_courant → g
|   |   boucle tantque eg.niveau > eg_cible.niveau
|   |   |   m coupé,eg_cible.niveau → état_géniteur[g]
|   |   |   m précédent[g] → g
|   |   |   m état_géniteur[g] → eg
```

Le nom n désigne un niveau, c'est-à-dire qu'il comporte la référence d'un géniteur non mort. Il faut détruire les strates en haut de la pile de sauvegarde qui sont postérieures à la strate du géniteur cité. Sur la figure 19, ce géniteur est en C11.
1 a pour rôle de traiter la situation du processus de récupération, pour le cas où il serait en cours de marquage d'une strate destinée à être détruite. En **2**, le processus d'exécution met à **coupé** tous les géniteurs jusqu'au niveau visé exclu, et réajuste leur niveau à celui visé, qui vaut 1 dans l'exemple de la figure 19.

```
mécanisme réduire( n1,n2,n3 :nom ) :<nom,nom,nom>
recherche_représentant(n1) → nn1
recherche_représentant(n2) → nn2
recherche_représentant(n3) → nn3
nn1,nn2,nn3 → résultat
1 démarrage_fournée(géniteur_courant,nn1,nn2,nn3) → géniteur_courant
```

Les représentants des trois noms passés en paramètre sont rendus en résultat. Ainsi sur la figure 20 les noms **c1**:C3, **niv**:C10 et **d1**:a sont respectivement des équivalents directs des noms **v1**:C0, **niv**:C11 et **d1**:a. Ces trois noms résument les accès dont l'utilisateur dispose désormais dans le cadre de la désignation. Ces informations sont confiées à une tentative de démarrage de fournée en 1, qui sera effective si le processus de récupération est en attente à ce moment-là.

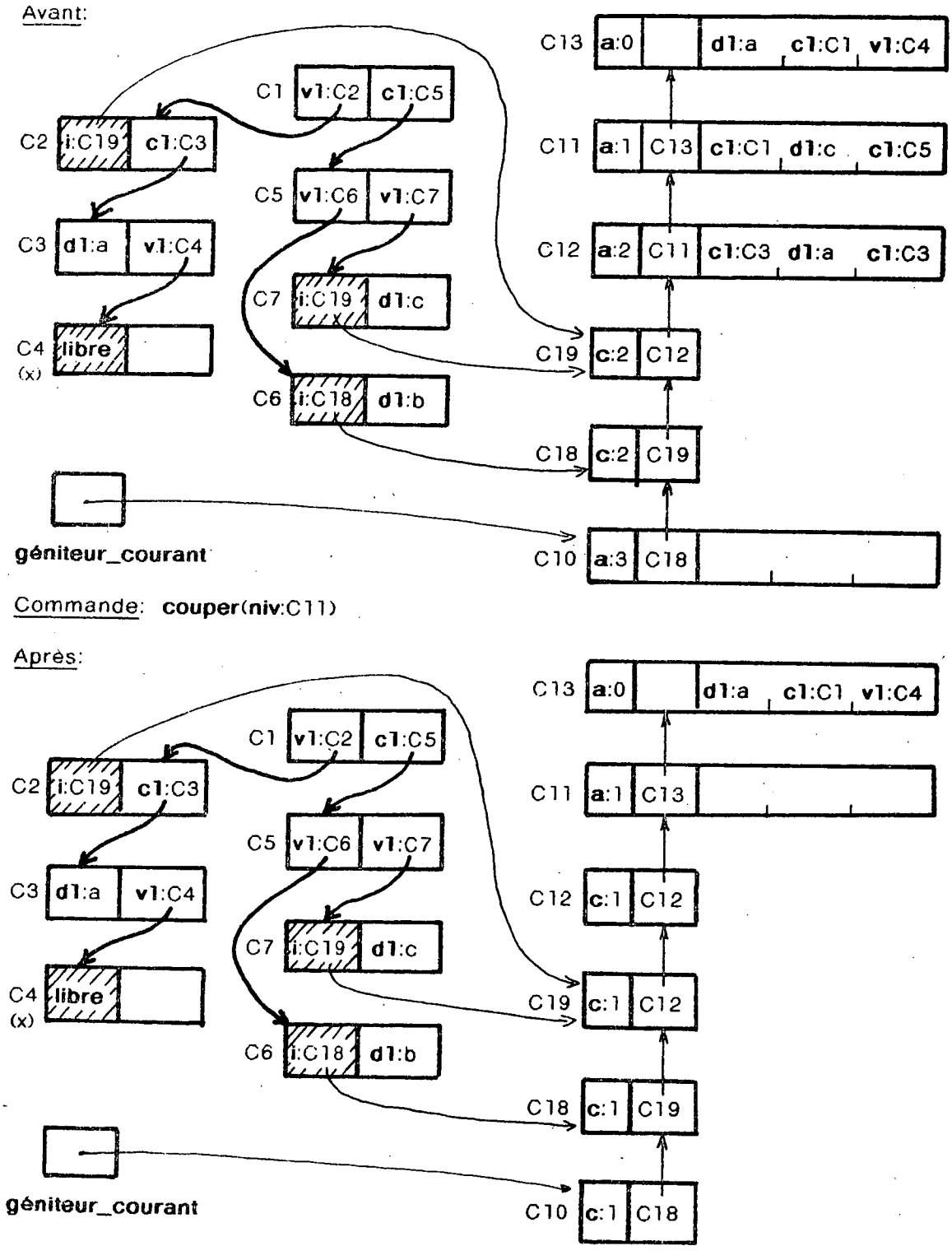


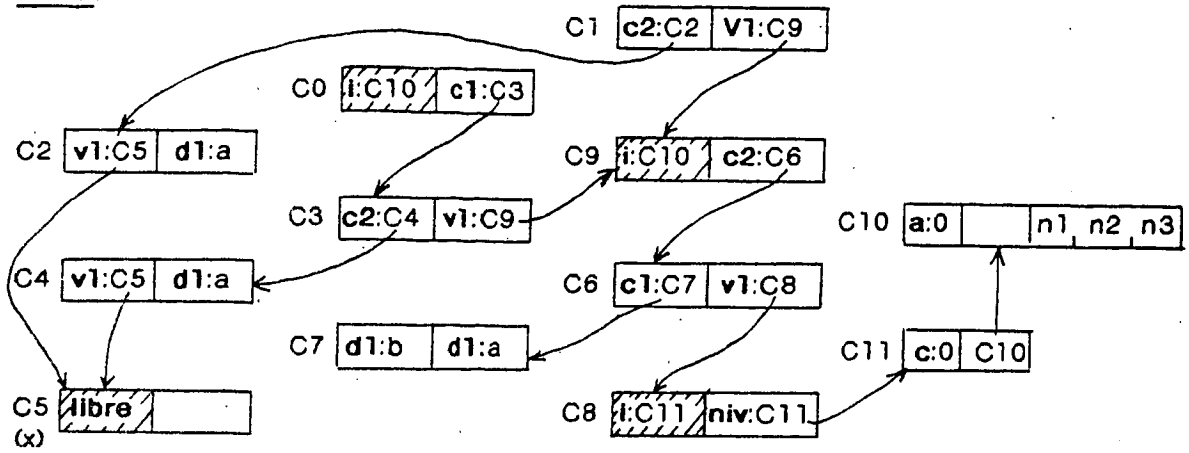
figure 19 - **couper**

Après exécution de cette commande la désignation offerte à l'utilisateur ne comporte que les trois noms directs rendus en résultat ainsi que des noms directs désignant les sous termes des termes accessibles par ces trois noms. Ceci est du à la propriété des commandes d'accès aux termes **accéder_gauche** et **accéder_droite** qui ne rendent en résultat que des noms directs. Ainsi sur la figure 20 les noms **v1**:C9 et **v1**:C8 ont disparu de la désignation bien que les cellules C8 et C9 participent toujours à la représentation des termes désignables après exécution de la commande. D'autre part les noms tels que **c1**:C1 et **c2**:C2 qui ne participent pas à la représentation des termes désignables après exécution de la commande, ont également disparu de la désignation.

mécanisme **initialiser**
initialisation_gestion_mémoire
allocation_cell_géniteur → g
m **actif**.0 → **état_géniteur**[g]
g → **géniteur_courant**

Un géniteur est créé. Il sera toujours actif et restera le plus bas dans la pile de sauvegarde. Son niveau est 0 et son champ **précédent** est sans signification.

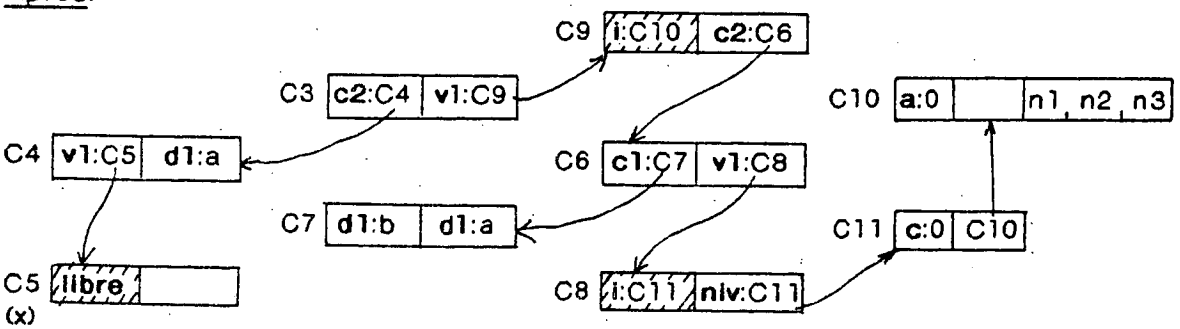
Avant:



D(c1:C1)=cons1(cons2(x,a),cons2(cons1(b,a),0))
D(c2:C2)=cons2(x,a)
D(c1:C3)=cons1(cons2(x,a),cons2(cons1(b,a),0))
D(v1:C9)=cons2(cons1(b,a),0)
D(c2:C4)=cons2(x,a)
D(v1:C5)=x
D(c2:C6)=cons2(cons1(b,a),0)
D(c1:C7)=cons1(b,a)
D(v1:C8)=0
D(v1:C0)=cons1(cons2(x,a),cons2(cons1(b,a),0))
D(niv:C11)=0 D(d1:a)=a D(d1:b)=b

Commande: **réduire**(v1:C0, niv:C11, d1,a)

Après:



D(c1:C3)=cons1(cons2(x,a),cons2(cons1(b,a),0))
D(c2:C4)=cons2(x,a)
D(v1:C5)=x
D(c2:C6)=cons2(cons1(b,a),0)
D(c1:C7)=cons1(b,a)
D(niv:C10)=0 D(d1:a)=a D(d1:b)=b

Résultat: c1:C3, niv:C10, d1:a

figure 20 - **réduire**

PROCESSUS DE RECUPERATION

STRUCTURE DE DONNEES

LA MEMOIRE DES STATUTS D'ALLOCATION

Le type d'information **statut_d'allocation** a pour valeurs possibles {**fournée1**, **fournée2**, **disponible**}. Le processus de récupération utilise un tableau **statut_alloc** de mémoires de type **statut_d'allocation**. Ce tableau est indicé par une **référence**.

Commentaire:
A chaque cellule est associé un statut d'allocation. Si celui-ci est **disponible** alors la cellule est considérée comme étant inutilisées par le processus d'exécution. Les deux valeurs **fournée1** et **fournée2** sont les statuts d'allocation des cellules utilisées. Il y a deux valeurs afin de permettre le parallélisme entre le processus d'exécution, consommateur de cellules, et le processus de récupération, producteur de cellules disponibles.

LES REGISTRES

registre **niveau_marquage** :**niveau**
registre **géniteur_sommet** :**référence**
registre **nom1**,**nom2**,**nom3** :**nom**
registre **fournée_courante** :{**fournée1**,**fournée2**}
Commentaire:
Les registres **niveau_marquage** et **géniteur_sommet** définissent la position de la strate couramment sous marquage dans la pile de sauvegarde. Les registres **nom1**, **nom2**, **nom3** définissent les trois termes conservés dans la strate.
Une cellule est "marquée" si le contenu de son **statut_d'allocation**" égale celui du registre **fournée_courante**; on "marque" une cellule en rangeant dans son **statut_d'allocation**" le contenu du registre **fournée_courante**.

RECUPERATION, PRINCIPE DES FOURNEES

mécanisme **récupération**
marquage; **ramassage**; **attente_fournée**

La récupération de mémoire se fait selon le principe des fournées. Dans le cadre d'une fournée, le processus de récupération marque les cellules accessibles puis ramasse les cellules non marquées, c'est à dire les rend disponibles à nouveau.

Le principe des fournées peut être brièvement décrit ainsi: la requête **démarrage_fournée** décrite plus loin découpe le temps en fournées successives. A tout instant l'espace des cellules allouées est partitionné en deux espaces. L'espace confirmé contient les cellules allouées depuis le début de la fournée et les cellules visitées par le marquage en cours. L'espace non confirmé contient les autres cellules.

En fin de marquage, toutes les cellules de l'espace non confirmé sont rendues disponibles. C'est le ramassage.

Après ramassage, l'espace non confirmé est alors vide. Une nouvelle fournée de récupération peut commencer avec un nouvel espace confirmé vide et avec un nouvel espace non confirmé égal à l'ancien espace confirmé. Il y a basculement de fournée. Le registre **fournée_courante** permet l'identification de l'espace confirmé, alternativement par **fournée1** et **fournée2**.

Parfois, le processus de récupération est interrompu en cours de marquage et forcé par le processus d'exécution à recommencer son activité après que les registres qui définissent son travail aient été réajustés, sans qu'il y ait changement de fournée.

MARQUAGE

```
mécanisme marquage
boucle tantque niveau_marquage > 0
|  11 marquage_terme(nom1); marquage_terme(nom2); marquage_terme(nom3)
|  sauv 12
|  |  décrémentation_niveau_marquage
|  |  disjoint → situation_exécution
```

Ce mécanisme a pour tâche de marquer les cellules utiles à la représentation de l'état défini par **géniteur_sommet**, **niveau_marquage**, **nom1**, **nom2**, **nom3** au moment de son lancement. Ce mécanisme traite les niveaux de la pile de sauvegarde l'un après l'autre, à partir de celui déterminé par **géniteur_sommet**, dans l'ordre décroissant. On profite ainsi de la façon dont l'ensemble des cellules qui participent à la représentation des termes d'une strate recouvre partiellement l'ensemble des cellules qui participent à la représentation des termes des strates de niveau inférieur, pour limiter le parcours aux cellules non déjà marquées, comme il sera précisé à l'occasion de l'exposé du mécanisme **marquage_terme**.

En **11** sont marquées les cellules servant à la représentation des termes mémorisés par le niveau.

En **12** il y a passage au niveau inférieur. Le processus d'exécution devient certainement disjoint.

```
mécanisme marquage_terme( nn :nom )
nn → n;  nulle_part → sorte_endroit; init_pile_parcours
boucle parcours
|   boucle 1 descente tantque n.indicateur ∈{v1,v2,c1,c2}
|   |   n.information → rcell
|   |   test_si_marquée(rcell) → m
|   |   si m = vrai alors 11; sortie descente
|   |   cas n.indicateur
|   |   v1,v2 alors
|   |   |   121
|   |   |   |   normalisation_liaison(rcell) → statut
|   |   |   |   si statut = libre alors  sortie descente
|   |   |   122
|   |   |   |   liais → sorte_endroit;  rcell → rcell_endroit
|   |   |   |   m liaison[rcell] → n
|   |   |   |   empile(remonter,rcell)
|   |   c1,c2 alors
|   |   |   13
|   |   |   |   g_cons → sorte_endroit;  rcell → rcell_endroit
|   |   |   |   m gauche[rcell] → n
|   |   |   |   empile(descendre_droite,rcell)
|   2
|   |   cas  n.indicateur
|   |   v1,v2 alors  marquage_cellule(rcell)
|   |   d2 alors  marquage_cellule(n.information)
|   |   niv alors 23 normalisation_atome_niv(n.information,
|   |                                  sorte_endroit,rcell_endroit)
|   boucle remontée
|   |   dépile → drapeau,quoi_faire,rcell
|   |   si drapeau = pile_vide alors sortie parcours
|   |   si quoi_faire ≠ remonter alors sortie remontée
|   |   marquage_cellule(rcell)
|   4
|   |   d_cons → sorte_endroit;  rcell → rcell_endroit
|   |   m droite[rcell] → n
|   |   empile(remonter,rcell)
```

Ce mécanisme parcourt les cellules qui participent à la représentation du terme désigné par *nn* afin de les marquer. Ceci consiste à parcourir un arbre, dont les noeuds sont constitués par les cellules portant des variables liées ou des constructeurs, et les feuilles sont constituées par les variables libres et les atomes. Le parcours est effectué en descendant prioritairement à gauche.

Pour gérer ce parcours, il est utilisé une mémoire auxiliaire fonctionnant en pile associée aux mécanismes

init_pile_parcours qui initialise la pile à l'état vide.

empile(:{remonter,descendre_droite}, :référence) qui empile les paramètres, **dépile** :<{pile_vide,pile_non_vide}, {remonter,descendre_droite}, référence> qui rend en résultat les informations dépilées sauf si la pile est vide, auquel cas le résultat est pile_vide.

La descente à travers un noeud consiste à empiler la référence du noeud, avec l'indication de ce qu'il faudra faire à la remontée, lorsqu'on aura dépilé cette référence.

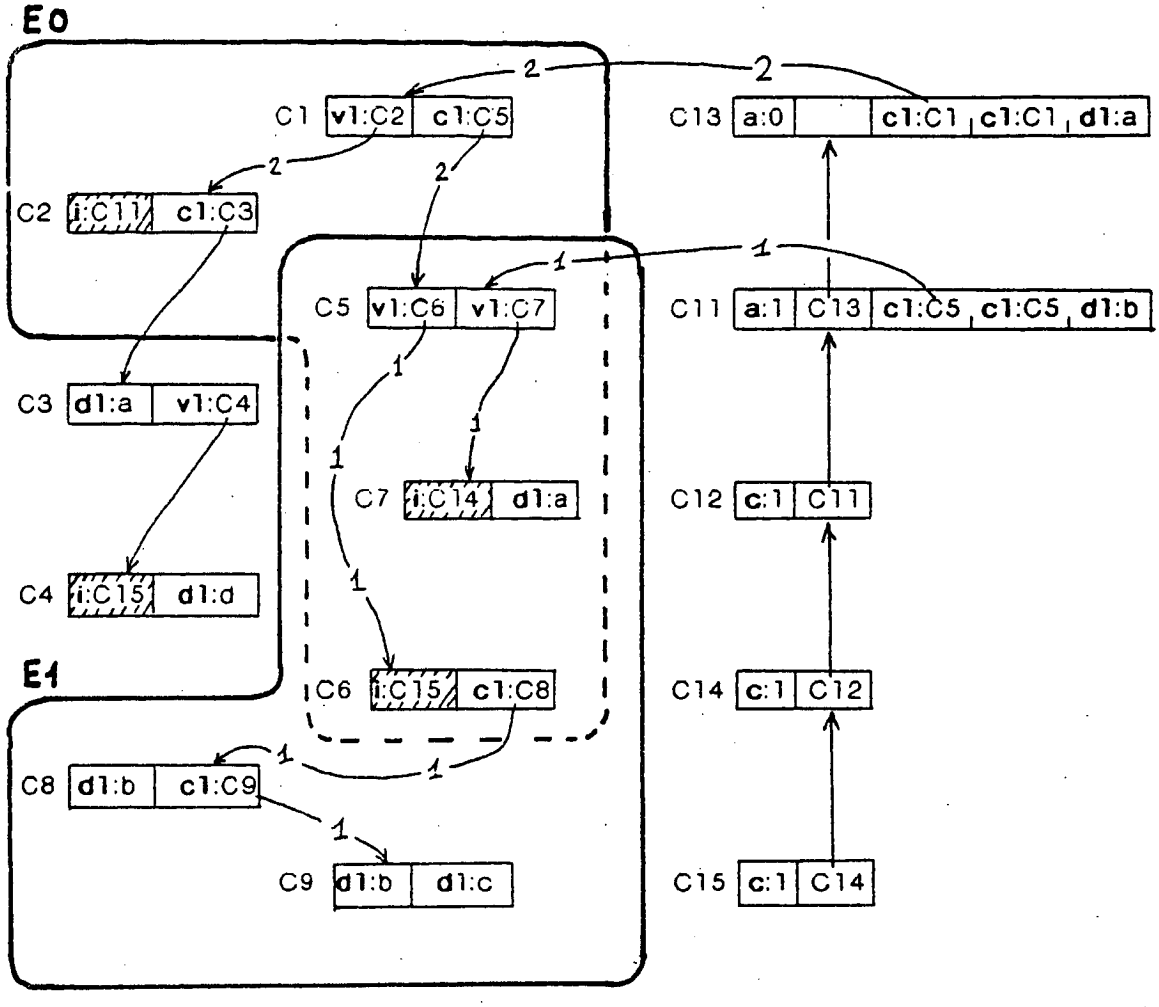


figure 21 - recouvrement des accès entre strates

Le marquage des noeuds se fait lors de la remontée, de sorte que si une cellule est marquée, alors toutes les cellules accessibles depuis cette cellule sont marquées. Cette propriété est nécessaire pour que les abandons de marquage provoqués par les mécanismes **recalage_sur_reprise** et **recalage_sur_coupure** puissent s'effectuer en laissant une situation de marquage cohérente.

1 Seuls les noms de variables ou de termes construits peuvent conduire le parcours à descendre, avec empilement de la référence de la cellule correspondante. Pour ces noms, il n'a servi à rien de se souvenir de l'endroit où se trouvait leur occurence, car elle n'aura pas à être mise à jour.

11 Le parcours tombe sur une cellule déjà marquée. Il ne descendra pas davantage. En effet, si une cellule est accessible dans le cadre de deux niveaux *i* et *j* tels que $i < j$, l'ensemble des cellules auquel la cellule donne accès dans le cadre de *i* est une partie de l'ensemble auquel elle donne accès dans le cadre de *j*: la différence de parcours venant exclusivement de la manière dont sont interprétées les cellules variables auxquelles cette cellule donne accès. Donc, à condition de parcourir les strates par niveaux décroissants, lorsqu'on rencontre une cellule qui a déjà été marquée lors du parcours, on peut faire l'économie du parcours des cellules auxquelles elle donne accès dans le cadre du niveau sous marquage, car elles sont certainement déjà marquées.
Ce phénomène est illustré sur la figure 21. Les ensembles de cellules E1 et E0 représentent respectivement les termes des strates de niveau 1 et 0 conservées dans les géniteurs C11 et C13. L'intersection de E1 et E0 contient les cellules C5, C6 et C7 qui représentent pour le niveau 1 les termes

cons1(**cons1**(b,**cons1**(b,c)),a),
a,
cons1(b,**cons1**(b,c)),

et qui représentent pour le niveau 0 les termes

cons1(x,y),
x,
y.

Pour chacune de ces cellules, le terme représenté par elle au niveau 1 est une instance du terme qu'elle représente au niveau 0.
Il se peut que le processus rencontre une cellule qui est marquée non pas parce que lui-même l'a déjà marquée lors de son parcours des strates, mais parce qu'elle a été allouée récemment, c'est-à-dire au cours de la fournée courante, au processus d'exécution. Le principe de ne pas parcourir les cellules auxquelles elle donne accès reste valable, car elle conduit à des cellules dont chacune est soit nouvellement allouée, donc automatiquement marquée, soit accessible au début de la fournée, et donc destinée à subir le marquage par ailleurs.

121 Le parcours a rencontré une cellule non encore marquée, portant une variable. Prise de décision s'il faut la considérer comme libre vue du niveau sous marquage, auquel cas il faut cesser de descendre, ou bien comme liée, auquel cas le parcours traverse la liaison et continue à descendre.

122 Traversée de la liaison. Le parcours s'attaque à la valeur de liaison de la variable et descend. Ceci se traduit par l'empilement de la référence à la variable. On doit aussi mémoriser à court terme l'endroit d'occurence du nom attaqué, c'est-à-dire la référence à la variable et l'indication qu'il s'agit d'une valeur de liaison, pour le cas où le nom serait de type niveau, ce qui ne sera découvert qu'au tour de boucle suivant.

Dans ce cas, on devra rectifier le contenu de l'endroit où le nom apparaissait, en y mettant un nouveau nom qui court-circuite les géniteurs coupés.

13 Le parcours a rencontré une cellule non encore marquée, portant un constructeur. Le parcours va descendre dans sa partie gauche, d'où l'empilement de la référence de la cellule, avec une directive qui servira, quand le parcours atteindra à nouveau cette cellule lors de la remontée, à décider de l'action à entreprendre, à savoir redescendre à droite plutôt que remonter encore. On mémorise aussi à court terme l'endroit d'occurence du nom dans la partie gauche du constructeur, pour la même raison qu'en **122**.

2 Diverses actions sont entreprises, correspondant à diverses sortes de feuilles du parcours.

23 Pour un atome niveau, il faut peut-être modifier le contenu de l'endroit où il apparait, pour lui faire citer le géniteur actif représentant un géniteur coupé. Ainsi, à la fin du marquage, il ne subsistera pas de références à des géniteurs coupés non marqués.

4 Préparation pour redescendre à droite.

```
mécanisme normalisation_liaison( ref :référence ) :{libre,liée}
m état_liaison[ref] → el
si el.statut=libre alors libre → résultat;     sortie
état_géniteur[el.géniteur] → étatgén
si étatgén.nature = mort alors
|   11   m libre.□ → état_liaison[ref]
|   libre → résultat; sortie
si étatgén.niveau > niveau_marquage alors 111;
|   libre → résultat; sortie
2
|   liée → résultat
|   si étatgén.nature = coupé alors 21
|   |   recherche_gén_représentant(el.géniteur) → g2
|   |   m indécis.g2 → état_liaison[ref]
```

Ce mécanisme décide si la variable portée par la cellule de référence *ref* est vue comme libre ou comme liée depuis le niveau cité par **niveau_marquage**. A cette occasion, il rectifie certaines informations dans le but de faire disparaître les références à des géniteurs morts ou coupés pour pouvoir ensuite récupérer la mémoire qu'ils occupent.

11 La variable est libre dans l'absolu. On normalise la situation en ramenant cette indication dans la cellule de la variable elle-même. Ceci permet de ramasser par la suite les cellules occupées par des géniteurs morts, puisqu'il n'y aura plus de références à eux. Ceci est illustré sur la figure 23.

111 La variable est liée, mais dans le cadre d'un niveau supérieur au niveau sous marquage; elle est donc considérée comme libre vue de ce niveau. Sa valeur de liaison n'est pas prise en considération, et le géniteur responsable de sa liaison n'est pas inspecté.

2 La variable est certainement liée vue du niveau sous marquage.

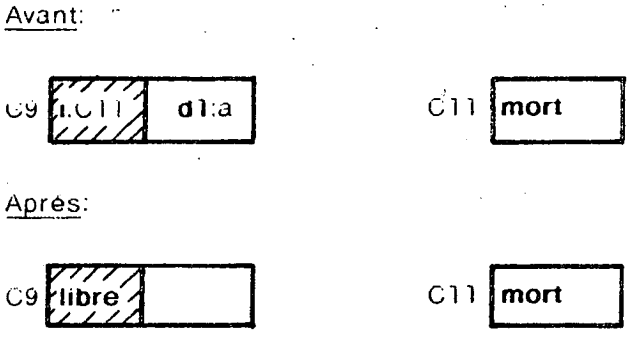


figure 23 - Normalisation de liaison pour une variable liée

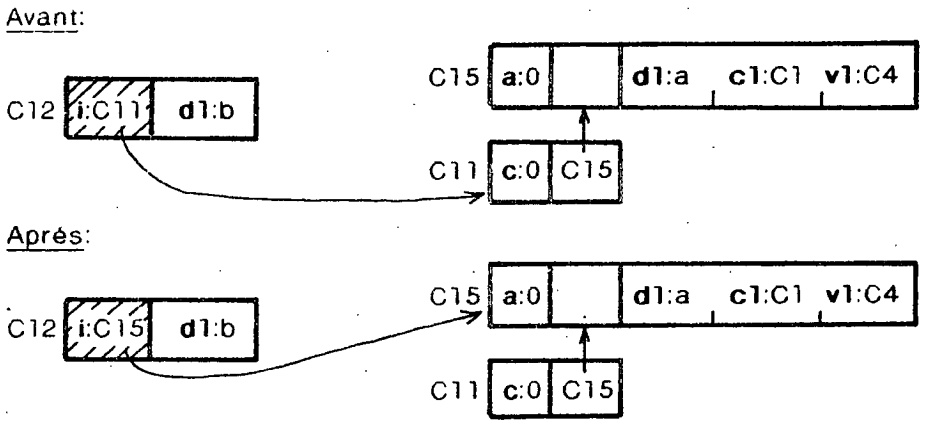


figure 24 - Normalisation de liaison pour une variable libre

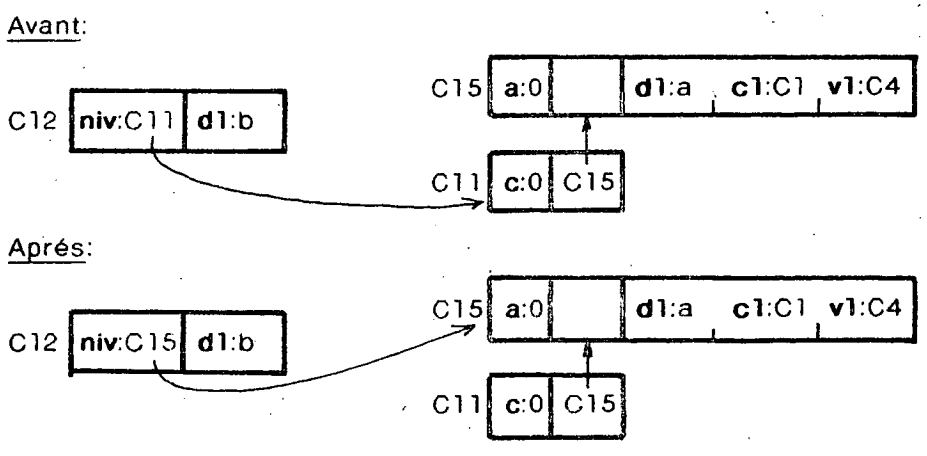


figure 25 - Normalisation d'un atome de type **niveau**

21 Normalisation de la référence depuis la variable au géniteur responsable de la liaison: cette référence est installée pour indiquer directement le géniteur actif correspondant au géniteur coupé. Ceci est illustré sur la figure 24. Grâce à cela, au fur et à mesure que le marqueur attaque les niveaux dans l'ordre décroissant, la disparition des références aux géniteurs coupés ayant un niveau inférieur au niveau attaqué permet, à la fin du traitement du niveau, de mettre à mort tous ses géniteurs coupés, et ultérieurement de les ramasser.

```
mécanisme normalisation_atome_niv( refg :référence,
          sorte_endroit :{g_cons,d_cons,liais}, rcell_endroit :référence )
m état_géniteur[refg] → étatgén
si étatgén.nature = coupé alors
|  recherche_gén_représentant(refg) → gr
|  cas sorte_endroit
|  g_cons alors niv.gr → gauche[rcell_endroit]
|  d_cons alors niv.gr → droite[rcell_endroit]
|  liais alors niv.gr → liaison[rcell_endroit]
```

refg est la référence d'un géniteur, *sorte_endroit* et *rcell_endroit* déterminent l'endroit où le mécanisme range la référence du premier géniteur actif antérieur à celui qui a *refg* pour référence. L'effet est illustré sur la figure 25.

```
mécanisme recherche_gén_représentant( g :référence ) :référence
g → g1
m état_géniteur[g1] → étatgén
boucle tantque étatgén.nature = coupé
|  m précédent[g1] → g1
|  m état_géniteur[g1] → étatgén
g1 → résultat
```

Ce mécanisme rend la référence du premier géniteur actif dans la chaine de géniteurs qui débute par le géniteur passé en paramètre.

```
mécanisme décrémentation_niveau_marquage
1
|  m état_géniteur[géniteur_sommet] → étatg1
|  boucle tantque étatg1.nature = coupé
|  |  m précédent[géniteur_sommet] → g1
|  |  m mort.□ → état_géniteur[géniteur_sommet]
|  |  g1 → géniteur_sommet
|  |  m état_géniteur[géniteur_sommet] → étatg1
2 marquage_géniteur(géniteur_sommet)
3
|  niveau_marquage - 1 → niveau_marquage
|  si niveau_marquage > 0 alors
|  |  31
|  |  |  géniteur_sommet → anté_g
|  |  |  m précédent[anté_g] → géniteur_sommet
|  |  |  recherche_gén_représentant(géniteur_sommet) → g
|  |  32
|  |  |  m g → précédent[anté_g]
|  |  |  m nom_1_2_3[g] → nom1,nom2,nom3
```

La figure 22 illustre la réorganisation du chainage des géniteurs et l'installation des registres qui définissent la nouvelle strate à marquer, lors du passage à un niveau inférieur.

1 Mise à mort des éventuels géniteurs coupés qui participent au niveau qui vient d'être marqué, jusqu'à rencontre du premier génteur actif, et marquage de ce dernier. Sur la figure, il est en C15.

3 Passage au niveau inférieur. A cette occasion, on peut détecter que la pile est épuisée.

31 Recherche du premier géniteur actif du niveau inférieur. Sur la figure, il est en C18

32 Décrochage de la chaine des éventuels géniteurs coupés qui aboutit au géniteur actif trouvé.

RAMASSAGE

```
mécanisme ramassage
0 → ref
boucle tantque ref ≤ référence_cellule_max
|   révision_statut_cellule(ref)
|   ref+1 → ref
```

Ce mécanisme parcourt toutes les cellules pour rendre disponibles celles qui portent encore la marque de la fournée précédente, ce qui est le signe qu'elles sont devenues inaccessibles à l'utilisateur et donc récupérables.

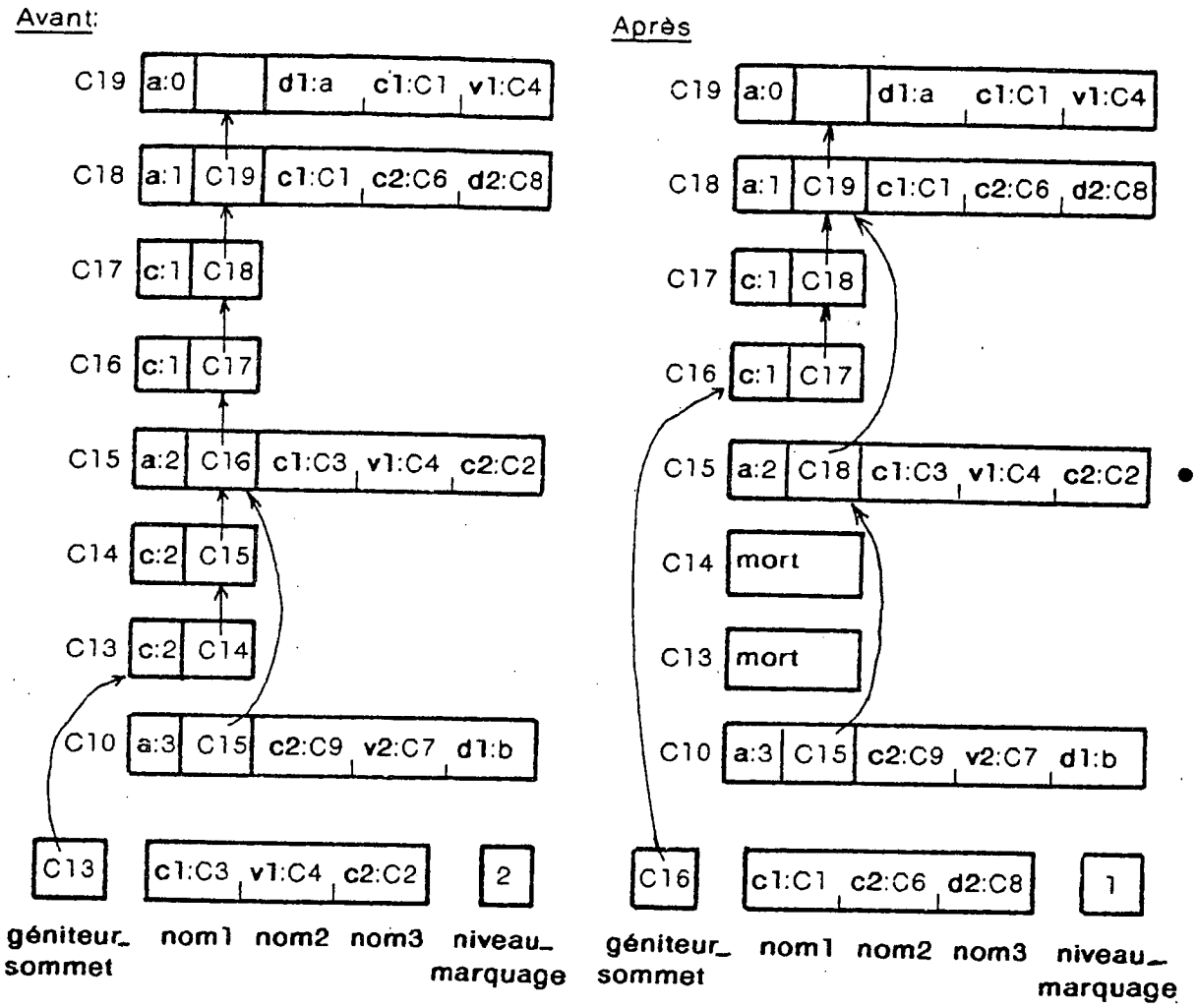
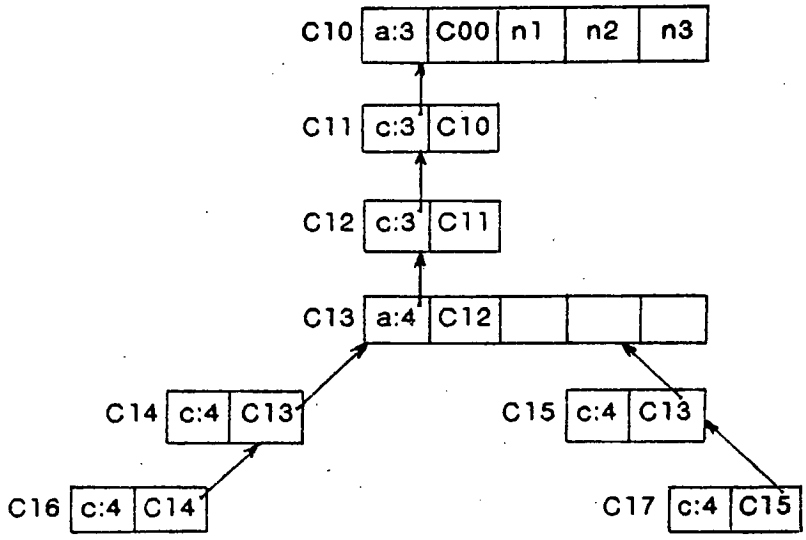


figure 22 - Réorganisation des géniteurs d'une strate

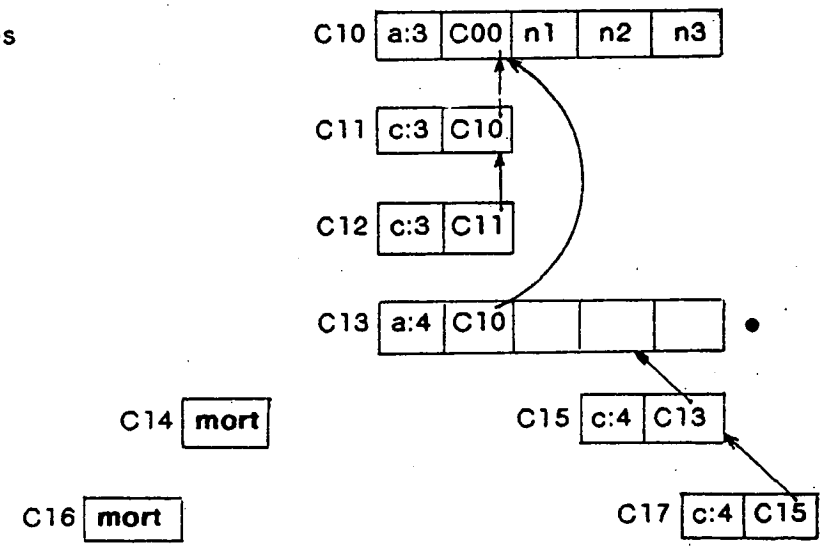
avant



géniteur_sommet = C16 **géniteur_courant** = C17

niveau_marquage = 4

après



géniteur_sommet = C12 **géniteur_courant** = C17
nom1, **nom2** et **nom1** = n1, n2 et n3
niveau_marquage = 3
situation_exécution = tangent

Le signe • signifie que la cellule à laquelle il est accolé, a été marquée pendant le déroulement du mécanisme illustré.

figure 26 - recalage_sur_reprise

MECANISMES CRITIQUES ET SYNCHRONISATIONS

Dans cette partie sont décrits les mécanismes qui accèdent aux mémoires partagées par les deux processus. Les phases critiques sont protégées par des verrous d'exclusion, déclarés sous les rubriques verrou. Les interactions se font par l'intermédiaire de conditions, déclarées sous les rubriques condition.

MODIFICATIONS DE LA PILE DE SAUVEGARDE PAR LE PROCESSUS D'EXECUTION

registre **situation_exécution** :{**tangent.disjoint**}

verrou *sauv*

Le registre **situation_exécution** sert à décider entre l'abandon ou la poursuite du marquage d'une strate détruite par une coupure. Lorsque la strate en cours de marquage a été, à un moment de son marquage, de même niveau que le géniteur courant, les processus sont dits "tangents". Sinon ils sont dits "disjoints". Dans le cas tangent, le processus de récupération ne peut pas abandonner, sous l'effet d'une coupure, le marquage de la strate en cours. En effet, dans ce cas, toutes les cellules accessibles depuis la strate sous marquage peuvent, au moment du passage à tangent, éventuellement participer à la désignation courante. En conséquence elles doivent toutes être marquées.

```
mécanisme recalage_sur_reprise( niv_visé :niveau )
sauv
|  si niv_visé ≤ niveau_marquage alors 1
|  |  si niv_visé < niveau_marquage alors 11
|  |  |  suspension récupération
|  |  |  décrémentation_niveau_marquage
|  |  |  recommencement récupération
|  |  12 tangent → situation_exécution
```

Ce mécanisme est illustré par la figure 26.

1 Le recalage n'est effectif que si le niveau repris est inférieur ou égal au niveau en cours de marquage. Dans ce cas des mesures sont à prendre pour assurer la cohérence des visions de la représentation de la part des deux processus.

11 Si le niveau cible de la reprise est strictement inférieur au niveau sous marquage, la strate en cours de marquage est détruite pour le processus d'exécution. Le processus de récupération est forcé à abandonner le marquage en cours et à entreprendre le marquage du niveau inférieur.

12 Dans le cas précédent ainsi que dans le cas où le niveau cible de la reprise est simplement égal au niveau sous marquage les deux processus deviennent tangents. En effet la strate qui est maintenant marquée est dans les deux cas reprise au titre de la désignation.

```
mécanisme recalage_sur_coupure( niv_cible :niveau )
sauv
|   si niveau_marquage ≥ niv_cible alors 1
|   |   suspension récupération
|   |   cas situation_exécution
|   |   disjoint alors 11
|   |   |   boucle tantque niveau_marquage ≥ niv_cible
|   |   |   |   décrémentation_niveau_marquage
|   |   |   recommencement récupération
|   |   tangent alors 12
|   |   |   géniteur_sommet → g; m état_géniteur[g] → egg
|   |   |   si egg.nature = actif alors vide → indic_liste_décrochée
|   |   |   sinon
|   |   |   |   non_vide → indic_liste_décrochée
|   |   |   |   boucle tantque egg.nature = coupé
|   |   |   |   |   m coupé.niv_cible → état_géniteur[g]
|   |   |   |   |   g → anté_décroché
|   |   |   |   |   m précédent[g] → g ; m état_géniteur[g] → egg
|   |   |   marquage_cell_géniteur(g) ; g → anté_g
|   |   |   boucle tantque egg.niveau > niv_cible
|   |   |   |
|   |   |   |   m précédent[g] → g ; m état_géniteur[g] → egg
|   |   |   |   si egg.nature = coupé alors
|   |   |   |   |   cas indic_liste_décrochée
|   |   |   |   |   vide alors
|   |   |   |   |   |   g → géniteur_sommet
|   |   |   |   |   |   non_vide → indic_liste_décrochée
|   |   |   |   |   non_vide alors m g → précédent[anté_décroché]
|   |   |   |   |   boucle tantque egg.nature = coupé
|   |   |   |   |   |   m coupé.niv_cible → état_géniteur[g]
|   |   |   |   |   |   g → anté_décroché
|   |   |   |   |   |   m précédent[g] → g ; m état_géniteur[g] → egg
|   |   |   |   m g → précédent[anté_g]
|   |   |   |   marquage_cell_géniteur(g) ; g → anté_g
|   |   |   niv_cible → niveau_marquage
|   |   |   continuation récupération
```

Ce mécanisme est illustré par la figure 27.

1 Des mesures sont à prendre si la coupure détruit la strate en cours de marquage et éventuellement des strates non encore marquées.

11 Dans le cas où les processus sont disjoints, le marquage en cours est abandonné et le processus de récupération est forcé à entreprendre le marquage de la strate juste inférieure à celles détruites par la coupure.

12 Dans le cas où les processus sont tangents, le marquage de la strate en cours ne peut pas être abandonné. Cependant pour donner au processus d'exécution une vision réelle de la liste des géniteurs cohérente avec sa vision logique, les géniteurs coupés de niveau supérieur ou égal au niveau cible de la coupure et datant d'avant le début de la fournée sont décrochés de cette liste.

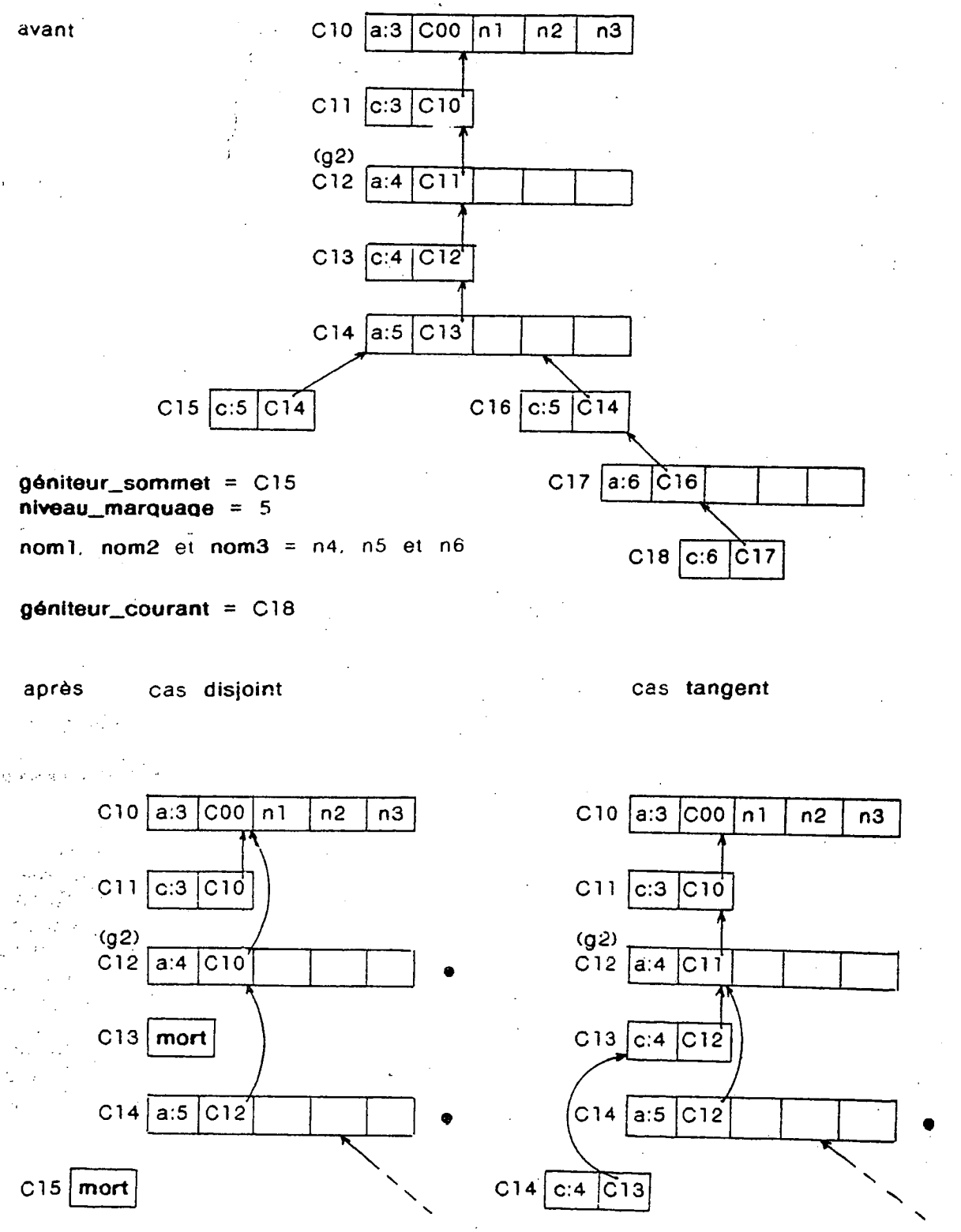


figure 27 - recalage_sur_coupure

SYNCHRONISATION POUR LE FONCTIONNEMENT PAR FOURNEES

```
registre récupération_en_attente :{vrai,faux}
verrou fournée
condition proposition_fournée

mécanisme démarrage_fournée( g1 :référence, n1,n2,n3 :nom ) :référence
fournée
|   cas récupération_en_attente
|   faux alors 1 g1 → résultat
|   vrai alors 2
|   |   21
|   |   |   cas fournée_courante
|   |   |   fournée1 alors fournée2 → fournée_courante
|   |   |   fournée2 alors fournée1 → fournée_courante
|   |   m état_géniteur[g1] → egg1
|   |   22
|   |   |   g1 → géniteur_sommet
|   |   |   egg1.niveau → niveau_marquage
|   |   |   n1 → nom1;  n2 → nom2;  n3 → nom3
|   |   23
|   |   |   recherche_gén_représentant(g1) → g2
|   |   |   g2 → résultat
|   |   24
|   |   |   tangent → situation_exécution
|   |   |   recommencement récupération
```

1 Si le processus de récupération n'est pas en attente, le mécanisme est ineffectif.

2 Si le processus de récupération est en attente, le démarrage d'une nouvelle fournée est effectif.

21 Inversion de la marque de fournée.

22 Initialisation des registres définissant le premier niveau à marquer.

23 La référence du géniteur actif représentant de *g1* est rendue en résultat. En effet les géniteurs coupés présents dans la pile de sauvegarde au début d'une fournée seront mis à mort et les cellules qu'ils occupaient récupérées à la fin de cette fournée. Logiquement, vue du processus d'exécution, la liste des géniteurs ne comporte plus que les géniteurs actifs. La chaine d'éventuels géniteurs coupés qui appartiennent à la strate qui va subir le marquage n'est plus connue que du marqueur, par le registre **géniteur_sommet**.

24 Lancement du processus de récupération au début de son mécanisme. Il y a tangence car la liste des sauvegardes vues par ce processus contient des informations utiles à la représentation de la désignation par le processus d'exécution.

```
mécanisme attente_fournée
fournée
|   vrai → récupération_en_attente
|   attente proposition_fournée
```

GESTION DE L'ALLOCATION DES CELLULES

```
registre consommateur_attend_cellule :{vrai,faux}
verrou alloc
condition cellule_disponible

mécanisme allocation_cellule :référence
alloc
|  boucle
|  |  recherche_cellule → r,m
|  |  si m ≠ trouvé alors 12
|  |  |  vrai → consommateur_attend_cellule
|  |  |  attente cellule_disponible
|  |  sinon faux → consommateur_attend_cellule ; sortie
|  2 fournée_courante → statut_alloc[r]
|  r → résultat
```

recherche_cellule est un mécanisme non décrit qui permet d'obtenir la référence d'une cellule disponible s'il en existe, ou la marque **non_trouvé** s'il n'en existe pas.

12 S'il n'existe pas de cellule au statut disponible, le processus d'exécution attend. Le prochain ramassage pourra le libérer de son attente.
2 La cellule rendue en résultat est marquée.

```
mécanisme révision_statut_cellule( r :référence )
alloc
|  statut_alloc[r] → a
|  si a ≠ fournée_courante et a ≠ disponible alors
|  |  restitution_cellule(a)
|  |  disponible → statut_alloc[r]
|  |  si consommateur_attend_cellule = vrai alors signal cellule_disponible
```

restitution_cellule est un mécanisme non décrit qui met à jour la gestion du stock des cellules disponibles; il est le symétrique de **recherche_cellule.**

```
mécanisme marquage_cellule( r :référence )
alloc fournée_courante → statut_alloc[r]

mécanisme marquage_cell_géniteur( r :référence )
```

Ce mécanisme, analogue à **marquage_cellule**, n'est pas décrit, pour ne pas entrer dans les détails sur la façon dont un géniteur est réparti sur plusieurs cellules.

```
mécanisme test_si_marquée( r :référence ) :{vrai,faux}
alloc
|  statut_alloc[r] → a
|  si a = fournée_courante alors vrai → résultat
|  sinon faux → résultat
```

MISE EN OEUVRE A BASE DE DEUX PROCESSEURS

La machine est mise en oeuvre sur un système comportant deux processeurs:

-le "processeur d'exécution" supporte les mécanismes non critiques du processus d'exécution, ainsi que le processus utilisateur qui sollicite les commandes,
-le "processeur de gestion de mémoire" supporte la totalité des mécanismes du processus de récupération, ainsi que les mécanismes critiques du processus d'exécution: **allocation_cellule**, **démarrage_fournée**, **recalage_sur_reprise** et **recalage_sur_coupure**.

STRUCTURE DE LA MACHINE

L'architecture de la machine est illustrée sur la figure 1

La "mémoire principale" est le réservoir des cellules de mémoire allouées dynamiquement. L'accès se fait au niveau de la demi-cellule, unité dans laquelle se cadrent bien les divers types d'information à conserver. Le nombre de cellules peut atteindre un million, une configuration normale étant de l'ordre de 128K. Leur largeur est de 48 bits. Le temps d'accès à une demi-cellule est de l'ordre de 400 ns. Elle peut être construite à base de circuits 4164 de la société Intel.

Le "processeur d'exécution", en interaction avec le processeur de gestion de mémoire, interprète les commandes du répertoire. Pour obtenir de bonnes performances, c'est un processeur micro-programmé. Une utilisation particulière de la machine, par exemple pour réaliser un interpréteur PROLOG, est réalisée par l'adjonction de micro-programmes complémentaires et de mémoires spécialisées, locales au processeur. Ce processeur peut être construit à l'aide de microprocesseurs en tranches de la série 2900 fournis par la société Advanced Micro Device. Le temps d'exécution d'une micro-instruction est de l'ordre de 150 ns.

Le "processeur de gestion de mémoire" assure la récupération des cellules de mémoire devenues inaccessibles à l'utilisateur. Ce processeur est responsable de la gestion du statut courant d'allocation des cellules conservé dans la mémoire des statuts d'allocation, à laquelle lui-seul accède. Il traite les requêtes émises par le processeur d'exécution. Ce processeur est de même technologie que le processeur d'exécution. Il dispose d'une mémoire privée de 16K mots pour contenir une pile, nécessaire à sa logique de fonctionnement.

La "mémoire des statuts d'allocation" comporte autant d'éléments que la mémoire principale a de cellules. Chaque élément code sur deux bits le statut d'allocation courant de la cellule qui lui correspond. Le temps d'accès à cette mémoire est de l'ordre de 100 ns. Elle pourra être construite à base de boîtiers Am9128-10 de la société Advanced Micro Device.

Le "bus mémoire" réalise et arbitre les accès des deux processeurs à la mémoire principale. Ce mécanisme est classique et peut être réalisé à l'aide des circuits développés par la société Intel pour le controle de son bus standard MULTIBUS.

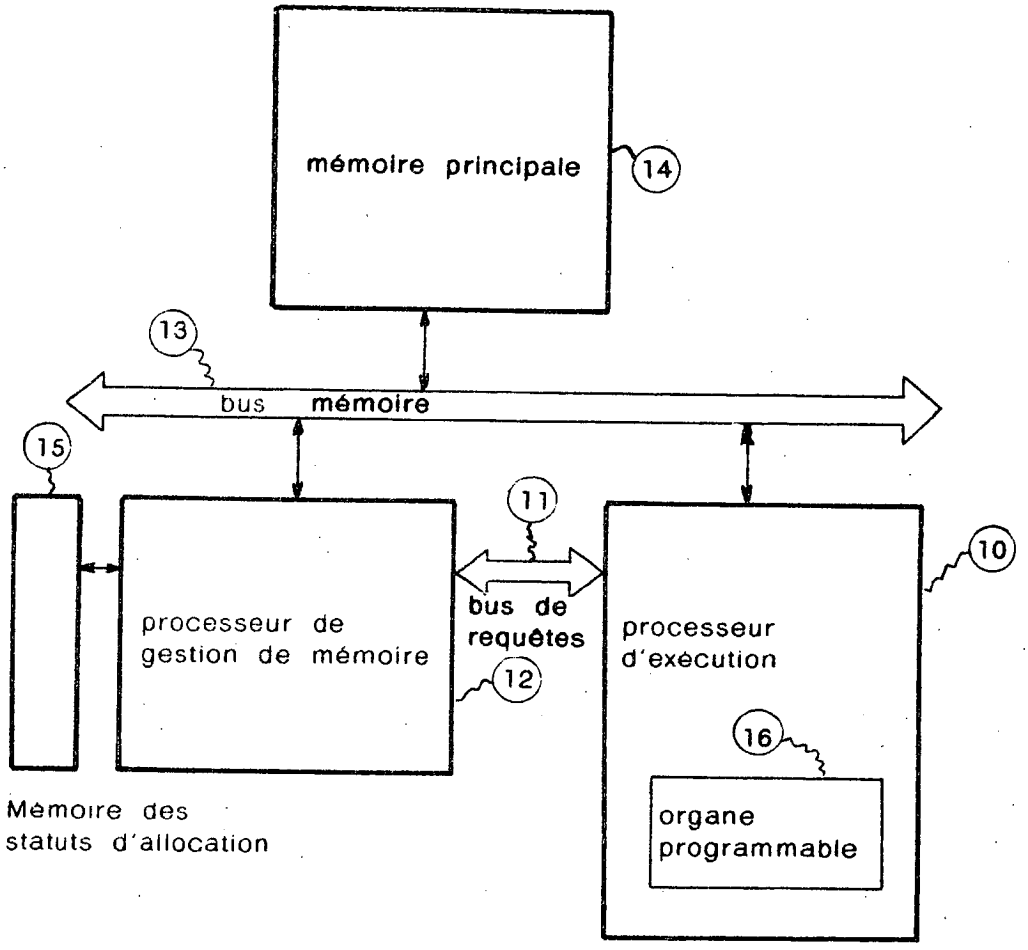


figure 1 - organisation de la machine

Le "bus de requêtes" assure l'échange d'information et la synchronisation entre les deux processeurs lors de certaines phases de leur activité afin que l'information contenue dans la machine reste globalement cohérente. Il consiste en quelques registres et des moyens d'échange de signaux.

REALISATION DES SYNCHRONISATIONS

Deux dispositifs matériels permettent de réaliser les synchronisations mentionnées dans les mécanismes précédemment décrits: ce sont
- un système de requêtes mis en oeuvre grâce au bus de requêtes.
- l'accès exclusif à la mémoire des cellules mis en oeuvre par le bus mémoire.

LE SYSTEME DE REQUETES

Le système de requêtes est un dispositif asymétrique qui permet au processeur d'exécution de forcer le processeur de gestion de mémoire à exécuter un mécanisme indiqué. Ceci permet d'une part d'accéder aux registres qui sont localisés sur le processeur de gestion de mémoire, et d'autre part assure l'exclusion avec le processus de récupération qui est interrompu le temps de l'exécution de la requête.

Le processeur d'exécution émet une requête en affichant sur le bus de requêtes l'identification du mécanisme à exécuter, accompagnée de ses paramètres. Le processeur d'exécution attend alors l'acquittement de la requête, accompagné des résultats affichées sur le bus de requêtes.

Le processeur de gestion de mémoire dispose de deux niveaux d'exécution des instructions: le niveau "de fond" et le niveau "requête". Deux opérations permettent de passer explicitement d'un niveau à l'autre:
- l'opération d'"invalidation" des requêtes pour passer au niveau requête.
- l'opération de "validation" des requêtes pour passer au niveau de fond.

Quand le processeur est au niveau de fond,
lors de la réception d'une requête, il sauvegarde son état de contrôle, passe automatiquement au niveau requête et exécute le mécanisme indiqué par la requête avec les paramètres fournis.

Quand le processeur est au niveau requête,
si une requête est reçue, elle n'est pas prise en compte immédiatement. Elle est mémorisée et sera prise en compte dès que le processeur repassera au niveau de fond.

Lors du traitement d'une requête,
l'opération d'"acquittement" débloque le processeur d'exécution en affichant les résultats sur le bus de requêtes.
l'opération "fin de requête" rétablit l'état de contrôle sauvegardé à la réception de la requête et fait repasser au niveau de fond.
l'opération de "forçage" détruit l'état de contrôle sauvegardé et provoque la poursuite de l'exécution en niveau de fond en un point de contrôle indiqué.

ACCES EXCLUSIF A LA MEMOIRE

Les deux processeurs accèdent à la mémoire de façon identique. Il n'y a qu'un seul bus d'accès à la mémoire et tous les accès sont donc séquentiels. Si un processeur demande un accès pendant que le bus est occupé à servir l'autre processeur, le premier processeur attend la fin du service en cours pour réaliser son accès. En cas de demandes simultanées, ce qui est assez probable car les deux processeurs sont finement synchrones pour des raisons de rapidité des échanges, le système de priorité câblé favorise le processeur d'exécution. Le service exclusif du bus mémoire peut être demandé pour un accès élémentaire unique ou pour une séquence quelconque d'accès. Dans ce dernier cas, le premier accès de la séquence est accompagné d'un ordre au bus mémoire de maintenir l'exclusion entre les accès ultérieurs, et le dernier accès de la séquence est accompagné d'un ordre de relâcher l'exclusion après cet accès.

Pour assurer l'exclusion d'accès à la mémoire de la part du processus de récupération, le processeur de gestion de mémoire réalise automatiquement l'invalidation des requêtes pendant qu'il est dans une séquence d'accès exclusif à la mémoire. Pour éviter les interblocages, le processeur d'exécution ne doit pas émettre de requête alors qu'il détient l'accès exclusif du bus mémoire. On constate que les mécanismes décrits respectent cette contrainte.

MISE EN OEUVRE DES DIVERSES SYNCHRONISATIONS LOGIQUES

Les interactions entre le processus d'exécution et le processus de récupération ont été décrites à l'aide de dispositifs de synchronisation purement logiques:

- verrous d'exclusion: *m, sauv, alloc, fournée*
- conditions d'attente: *cellule_disponible, proposition_fournée*
- actions sur un processus: suspension, continuation, recommencement.

Ces dispositifs logiques ont permis de décrire la réalisation en limitant le moins possible la diversité des architectures matérielles possibles. Dans le cas de l'architecture présentée la tâche de fond du processeur de gestion de mémoire correspond au processus de récupération. Les dispositifs logiques de synchronisation sont réalisés comme suit à l'aide des dispositifs matériels dont disposent les processeurs.

Le verrou *m* est directement réalisé par le maintient et le relâchement de l'exclusion d'accès à la mémoire.

Les verrous *sauv, alloc* et *fournée* sont réalisés par le sytème de requêtes. Ces verrous sont sollicités à l'occasion de certains mécanismes du processus d'exécution: **recalage_sur_reprise, recalage_sur_coupure, allocation_cellule, démarrage_fournée**. Ces mécanismes sont exécutés par requêtes sur le processeur de gestion de mémoire. Ceci permet, outre l'exclusion naturellement offerte par le système de requêtes, de profiter de la localité d'accès aux registres et à la mémoire des statuts d'allocation nécessaires à l'exécution de ces mécanismes. La pratique de ces verrous par le processus de récupération est simplement réalisée par les opérations d'invalidation et de validation des requêtes sur le processeur de gestion de mémoire.

La réalisation se comporte donc comme si ces trois verrous étaient le même. On sait que du point de vue logique on peut toujours transformer plusieurs verrous d'exclusion en un seul. Cependant, pour des raison d'efficacité, il serait préférable que le verrou *alloc*, qui est sollicité à chaque allocation de cellule, soit différencié. Cela serait possible avec un système de requêtes à plusieurs niveaux, le niveau le plus prioritaire étant utilisé pour les allocations de cellule.

La condition *cellule_disponible* est attendue par le processus d'exécution, à l'occasion du mécanisme **allocation_cellule** qui est exécuté par requête sur le processeur de gestion de mémoire. Cette attente est réalisée en terminant la requête d'allocation par une simple fin de requête, sans acquittement de la requête. Le signalement de cette condition par le processus de récupération a lieu lorsqu'il est en phase de ramassage et se traduit par un acquittement retardé de cette requête.

La condition *proposition_fournée* est attendue par le processus de récupération. Cette attente est réalisée par une boucle vide exécutée au niveau de fond sur le processeur de gestion de mémoire.

Les actions de suspension, continuation et recommencement ne portent que sur le processus de récupération. Elles sont causées par les mécanismes **recalage_sur_reprise** et **recalage_sur_coupure** qui sont exécutés au niveau requête sur le processeur de gestion de mémoire. La suspension est automatiquement réalisée par la prise en compte de la requête. La continuation est réalisée par l'opération de fin de requête qui restaure l'état de contrôle sauvegardé au début de la requête. Le recommencement est réalisé par une opération de forçage avec pour point de contrôle le début du mécanisme de récupération.

Récentes publications de l'IRISA

PI 200 **Etude générale d'un réseau constitué de deux stations hyperexponentielles**
Jean-Yves Le Boudec , 12 pages ; Mai 1983

PI 201 **Langage de Dyck et groupe symétrique**
Yves Cochet , 13 pages ; Juin 1983

PI 202 **On the observational semantics of pair parallelism**
Philippe Darondeau, Laurent Kott , 14 pages ; Juin 1983

PI 203 **Les langages fonctionnels : caractéristiques, utilisation et mise en œuvre**
Daniel Le Métayer , 162 pages ; Juin 1983

PI 204 **On exact and approximate itterative methods for general queueing networks**
Raymond A. Marie, William J. Stewart , 27 pages ; Juin 1983

PI 205 **On quantifier hierarchy and its paraphrase in a semantic representation of natural language sentences**
Patrick Saint-Dizier , 17 pages ; Juillet 1983

PI 206 **Trois articles sur le traitement adaptatif du signal pour l'encyclopédie Pergamon sur l'automatique**
Albert Benveniste , 64 pages ; Septembre 1983

PI 207 **Sur l'existence et l'unicité du réseau homogène à un réseau ferme de files d'attente à lois générales**
Raymond Marie, Gérardo Rubino , 44 pages ; Septembre 1983

PI 208 **Problèmes d'implémentation du langage Prolog en vue de la réalisation d'une machine Prolog (Rapport final ATP « Intelligence Artificielle »)**
Yves Bekkers, Bernard Canet, Olivier Ridoux, Lucien Ungaro , 63 pages ; Octobre 1983

PI 209 **La technique du suivi de contour en synthèse d'images et ses applications**
Gérard Hégron , 28 pages ; Octobre 1983

PI 210 **A new characterization of infinitary rational languages**
Philippe Darondeau, Laurent Kott , 9 pages ; Octobre 1983

PI 211 **On the observational semantics of fair parallelism**
Philippe Darondeau, Laurent Kott , 80 pages ; Octobre 1983

PI 212 **Solution à forme produit d'un système linéaire**
J. Pellaumail , 36 pages ; Novembre 1983

PI 213 **Equations de Chapman-Kolmogorov et flots stationnaires pour des processus markoviens**
J.Y. Le Boudec et J. Pellaumail , 18 pages ; Novembre 1983

PI 214 **Distribution des interentrées et intersorties pour des réseaux à forme produit**
J.Y. Le Boudec , 39 pages ; Novembre 1983

PI 215 **Un outil informatique pour l'analyse graphique de données**
René Thoraval , pages ; Juin 1983

PI 216 **Perturbation de la décomposition spectrale d'une matrice hermitienne**
Bernard Philippe , 21 pages ; Novembre 1983

PI 217 **L'IRISA vu à travers les stages effectués par ses étudiants de DEA (1ère année de thèse)**
Edition 1983-1984 , Daniel Herman ; 28 pages Janvier 1984

PI 218 **Contribution de la classification automatique pour l'organisation et l'interrogation d'un corpus de « petites annonces »**
Philippe Peter , 32 pages ; Janvier 1984

PI 219 **A micro-computer implementation of an interactive functional programming system**
Wei Zi Chu , 22 pages ; Janvier 1984

PI 220 **Commande en boucle fermée de robots munis de capteurs extéroceptifs terminaux**
Bernard Espiau , 81 pages ; Janvier 1984

PI 221 **Justification et validité statistique d'une échelle** [0,1] de fréquence mathématique pour une structure de proximité sur un ensemble de variables observées
I.C. Lerman , 48 pages ; Janvier 1984

PI 222 **Spécification d'une machine de gestion mémoire pour les interpréteurs des langages logiques - version 1 (provisoire)**
Yves Bekkers, Bernard Canet, Olivier Ridoux, Lucien Ungaro , 82 pages ; Février 1984

PI 223 **Une approche à la validation des protocoles d'Enchère par la méthode des tests de spécification**
Christine Ecault , 36 pages ; Février 1984

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

IRIA
laboria

Institut de Recherche
d'Informatique
et d'Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105 78150 - Le Chesnay
France
Tél. 954 90 20

laboratoire de recherche
en informatique
et automatique

ON
THE UNIFORM HALTING PROBLEM
FOR
TERM REWRITING SYSTEMS

Gérard HUET
Dallas LANKFORD

Rapport de Recherche N° 283

Mars 1978

8p.

ON THE UNIFORM HALTING PROBLEM FOR TERM REWRITING SYSTEMS

Gérard Huet*, Dallas Lankford**

Résumé :

On montre que le problème de l'arrêt uniforme des systèmes de réécriture de termes est indécidable de degré 0'', même en se restreignant aux symboles de fonction monadiques. On montre que par contre ce problème est décidable pour les termes sans variables.

Abstract :

We show that the uniform halting problem for term rewriting systems is undecidable of degree 0'', even when terms are restricted to monadic function symbols. We also show that the uniform halting problem for ground term rewriting systems is decidable.

* IRIA/LABORIA
** USC - ISI

1. Introduction

Let T be the set of terms of a first-order logic. A term rewriting system over T is a finite set :

$$R = \{<\gamma_i, \delta_i> \mid 1 \leq i \leq N\} \text{ with } \forall i \leq N \; V(\delta_i) \subset V(\gamma_i)$$

where $V(t)$ denotes the set of variables appearing in term t.

We define, in the same way as in [2], the relation $\xrightarrow[R]{}$ over T as the smallest relation containing R and closed by :

a) for every substitution σ $\quad t \xrightarrow[R]{} t' \Longrightarrow \sigma(t) \xrightarrow[R]{} \sigma(t')$

b) $t_i \xrightarrow[R]{} t_i' \Longrightarrow Ft_1 \cdots t_n \xrightarrow[R]{} Ft_1 \cdots t_{i-1} t_i' t_{i+1} \cdots t_n$

for any function symbol F of arity n.

In the following we abbreviate $\xrightarrow[R]{}$ by $\rightarrow$.

A term t is said to be *immortal* in R iff there exists an infinite sequence starting from t :

$$t = t_0 \rightarrow t_1 \rightarrow \cdots t_n \rightarrow \cdots .$$

If no term t is immortal in R, we say that R is *noetherian*. The uniform halting problem for R is to determine whether or not R is noetherian.

In order to do that, we shall show how to map this problem into the uniform halting problem for Turing machines.

2. The construction of R_M

Let M be any Turing machine, with

input alphabet $\Sigma = \{s_0, s_1, \cdots, s_n\}$ and

state alphabet $Q = \{q_0, q_1, \cdots, q_p\}$.

We assume $s_0 = *$, the blank symbol.

We consider the set T_M of terms built from the following function symbols :

$$F_1 = \{\overleftarrow{s}_0, \cdots, \overleftarrow{s}_n, \overrightarrow{s}_0, \cdots, \overrightarrow{s}_n, Q_0, \cdots, Q_p, L\}$$

and $F_0 = \{R\}$. All symbols in F_1 have arity 1, R has arity 0. We shall write the terms in T_M as strings over $F_1^*(F_0 \cup \{x\})$.

Let I be an instantaneous description of machine M :

$$I = <w_1, q_i, w_2>$$

where $w_1 \in \Sigma^*$ is the contents of the tape to the left of the head, $w_2 \in \Sigma^*$ is the

contents of the tape to the right of the head, q_i is the current state (we assume, as usual, the tape blank everywhere except in a finite portion). We code I by the term :

$$t_I = L\ \vec{w}_1\ Q_i\ \overleftarrow{w}_2\ R \in \mathcal{T}_M.$$

Example : With $I = \langle s_1 s_2, q_2, s_1 s_4 \rangle$,
corresponding to the configuration :

...	*	*	s_1	s_2	s_1	s_4	*	*	...
				↑ q_2					

we get : $$t_I = L\ \vec{s}_1\ \vec{s}_2\ Q_2\ \overleftarrow{s}_1\ \overleftarrow{s}_4\ R.$$

Let us now show how to code the program of M with a term rewriting system $\mathcal{R}_M$ over $\mathcal{T}_M$.

For any right-moving instruction of M :
"in state q_i reading s_j, write s_k and go right in state q_ℓ"
we put in $\mathcal{R}_M$ the rule $\langle Q_i\ \overleftarrow{s}_j\ x,\ \vec{s}_k\ Q_\ell\ x \rangle$.

If j=0 we add also the rule :

$$\langle Q_i R,\ \vec{s}_k\ Q_\ell R \rangle ,$$

corresponding to reading a new portion of tape to the right.

For any left-moving instruction of M :
"in state q_i reading s_j, write s_k and go left in state q_ℓ"
we put in $\mathcal{R}_M$ the rules :

$$\langle \vec{s}_m\ Q_i\ \overleftarrow{s}_j\ x,\ Q_\ell\ \overleftarrow{s}_m\ \overleftarrow{s}_k\ x \rangle \text{ for all m, } 0 \le m \le n,$$

and $$\langle L\ Q_i\ \overleftarrow{s}_j\ x,\ L\ Q_\ell\ \overleftarrow{s}_0\ \overleftarrow{s}_k\ x \rangle.$$

If j=0 we add also the rules :

$$\langle \vec{s}_m\ Q_i\ R,\ Q_\ell\ \overleftarrow{s}_m\ \overleftarrow{s}_k\ R \rangle \text{ for all m, } 0 \le m \le n,$$

and $$\langle L\ Q_i\ R,\ L\ Q_\ell\ \overleftarrow{s}_0\ \overleftarrow{s}_k\ R \rangle.$$

The construction of $\mathcal{R}_M$ is such that

$$I \xrightarrow[M]{} I' \iff t_I \xrightarrow[\mathcal{R}_M]{} t_I'.$$

From this follows immediately :

Lemma 1 : If the instantaneous description I of M is immortal, then the term t_I is immortal in $\mathcal{R}_M$.

(We recall that an instantaneous description of a Turing machine M is immortal iff starting with it M will never halt).

Corollary

The problem of determining, given R and t, whether t is immortal in R, is undecidable of degree $0'$.

Proof :

The halting problem for M on input x reduces to determining whether t_I is immortal in R_M, where I is the initial instantaneous description of M with tape x. □

Note that the converse of lemma 1 holds also. However, this is too weak to give an answer to the uniform halting problem, since there are terms in T_M which do not code an instantaneous description. The next section will show how to get a stronger converse.

3. Equivalence of the uniform halting problems of M and R_M

Let us introduce the notation :

$$\vec{S} = \{\vec{S}_0, \cdots, \vec{S}_n\}$$

$$\overleftarrow{S} = \{\overleftarrow{S}_0, \cdots, \overleftarrow{S}_n\}$$

$$Q = \{Q_0, \cdots, Q_p\} .$$

Lemma 2 : If a term t is immortal in R_M, M possesses some immortal instantaneous description.

Proof :

Any word t in T_M may be written as :

$$t = u_1 \, v_1 \, u_2 \, v_2 \cdots v_q \, u_{q+1} \, R \qquad q \geq 0$$

with $\quad u_i \in (\vec{S} \cup \overleftarrow{S} \cup \{L\})^* \qquad i \leq q+1$

and $\quad v_i \in \vec{S}^* \, Q \, \overleftarrow{S}^* \qquad i \leq q.$

We assume furthermore that the v_i's are maximal, i.e. u_i does not end with an $\vec{S}_k$ $(1 \leq i \leq q)$, and u_i does not begin with an $\overleftarrow{S}_k$ $(2 \leq i \leq q+1)$.

Claim : If $t \xrightarrow[R_M]{} t'$ then $q>0$ and there exists a j, $1 \leq j \leq q$ such that

- $t' = u_1 v_1 \cdots u_j\ v'_j u_{j+1} \cdots v_q u_{q+1}\ R$,
- $Lv_j R \xrightarrow[R_M]{} Lv'_j R$
- v'_j is of the form above.

This claim is easy to establish, by cases on the rule used to derive t into t'.

Now, if t is immortal, there must exist a $j \leq q$ whose v_j is rewritten infinitely often, and therefore such that $Lv_j R$ is immortal in R_M. $Lv_j R$ is the code of an instantaneous description of M which is immortal by lemma 1. □

Combining lemma 1 and lemma 2, we get that R_M is noetherian iff M has no immortal instantaneous description. This second problem being undecidable of degree 0", using the results in Herman [1], we get :

Theorem 1

The uniform halting problem for term rewriting systems is undecidable of degree 0", even for terms restricted to monadic function symbols.

Remark :

In [3] it is claimed, without proof, that the uniform halting problem is undecidable, even when $|R| \leq 3$.

We shall show in the next section that the uniform halting problem is decidable for ground term rewriting systems.

4. Ground term rewriting systems

We shall now restrict ourselves to the case where T consists only of *ground terms*, i.e. with no variables. A ground term rewriting system is a finite set :

$$R = \{\langle \gamma_1, \delta_i \rangle \mid 1 \leq i \leq N\} \quad \forall i \leq N \quad V(\gamma_i) = V(\delta_i) = \emptyset.$$

Lemma 3 :

If t is immortal in R, there exists $i \leq N$ such that δ_i is immortal in R.

Proof : By induction on t.

Let $t = t_0 \to t_1 \to t_2 \to \cdots$ be an infinite $\to$-sequence, and let $u_i \in N_+^*$ be the occurrence of t_i which is reduced at step i.

If $\exists k \geq 0$ such that $u_k = \Lambda$ (the null, or top occurrence) then $t_{k+1} = \delta_k$ is immortal.

Otherwise, the sequence is internal, and let F be the common leading function symbol of all t_i's, p its arity : $t_i = Ft_i^1 \cdots t_i^p \quad \forall i \geq 0$. For every j, $1 \leq j \leq p$, let $I_j = \{i \mid u_i \text{ begins with } j\}$. At least one of the I_j's is infinite, let us say I_k : $I_k = \{k_1, k_2, \cdots\}$ with $\forall i \geq 0 \quad 0 \leq k_i \leq k_{i+1}$.

Then the subsequence

$t_0^k = t_{k_1}^k \to t_{k_2}^k \to \cdots$ is an infinite $\to$-sequence, and the result follows by induction hypothesis. □

Lemma 4 :

If $\exists i \leq N$ such that δ_i is immortal in R, then $\exists j \leq N$ such that $\exists t \ \exists u \in O(t) \ \gamma_j \overset{*}{\to} t \ \& \ t/u = \gamma_j$.

Proof : By induction on $N = |R|$.

- If $N=0$ this is trivially true.
- Otherwise, assume δ_i is immortal in R :

$$\delta_i = t_0 \to t_1 \to t_2 \to \cdots$$

If the i-th rule is used in this reduction, then we can take j=i.

Otherwise, δ_i is immortal in $R' = R - \{<\gamma_i,\delta_i>\}$, and, using lemma 3, there exists $<\gamma_k,\delta_k> \in R'$ such that δ_k is immortal in R'. By induction hypothesis, there exists $<\gamma_j,\delta_j> \in R' \subset R$ answering the condition. □

Theorem 2 :

The uniform halting problem for ground term rewriting systems is decidable.

Proof :

Let R be a ground term rewriting system. Either R is noetherian, in which case all reductions from a given term terminate, or else using lemmas 3 and 4 there exists a rule $<\gamma,\delta> \in R$ such that δ reduces to a term containing γ as a subterm. We can therefore decide the uniform halting problem for R by enumerating, level by level, all the reductions of the right sides δ's of R, checking for the occurrence of the corresponding left side γ. Note that these reduction trees are finitely branching, R being finite. □

Bibliography

[1] G. Herman
Strong computability and variants of the uniform halting problem.
Z. Math. Logik Grundl. Math., 17 (1971), 115-131.

[2] G. Huet
Confluent reductions : abstract properties and applications to term rewriting systems.
Proceedings of 18th annual IEEE Symposium on Foundations of Computer Science, Oct. 77.

[3] R. Lipton & L. Snyder
On the halting of tree replacement systems.
Conference on theoretical computer science, University of Waterloo, July 1977.